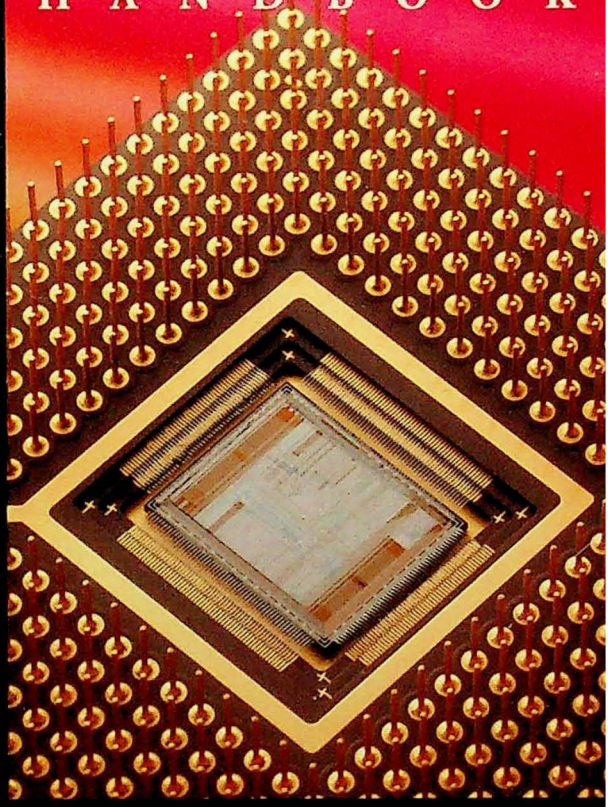
Alpha Architecture HANDBOOK



ital

Alpha Architecture Handbook

digital

Digital believes that the information in this publication is accurate as of its publication date; such information is subject to change without notice. Digital is not responsible for any inadvertent errors.

Copyright © 1992 Digital Equipment Corporation

All rights reserved. Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation: PDP-11, VAX, VMS, ULTRIX, and the Digital logo.

OSF/1 is a registered trademark of Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Table of Contents

Preface xi

Chapter 1 • Introduction

The Alpha Approach to RISC Architecture 1-1
Data Format Overview 1-3
Instruction Format Overview 1-4
Instruction Overview 1-4
Instruction Set Characteristics 1-6
Terminology and Conventions 1-6
Numbering 1-6
Security Holes 1-6
UNPREDICTABLE and UNDEFINED 1-7
Ranges and Extents 1-8
ALIGNED and UNALIGNED 1-8
Must Be Zero (MBZ) 1-8
Read As Zero (RAZ) 1-8
Should Be Zero (SBZ) 1-8
Ignore (IGN) 1-8
Implementation Dependent (IMP) 1-9
Figure Drawing Conventions 1-9
Macro Code Example Conventions 1-9

Chapter 2 • Basic Architecture

Addressing 2-1
Data Types 2-1
Byte 2-1
Word 2-1
Longword 2-2
Quadword 2-2
VAX Floating-Point Formats 2-3
F_floating 2-3
G_floating 2-4
D_floating 2-5
IEEE Floating-Point Formats 2-6
S_Floating 2-7
T_floating 2-8
Longword Integer Format in Floating-Point Unit 2-9
Quadword Integer Format in Floating-Point Unit 2-10
Data Types with No Hardware Support 2-11

Chapter 3 • Instruction Formats

Alpha Registers 3-1
Program Counter 3-1
Integer Registers 3-1
Floating-Point Registers 3-2
Lock Registers 3-2
Optional Registers 3-2
Memory Prefetch Registers 3-2
VAX Compatibility Register 3-2
Notation 3-2
Operand Notation 3-3
Instruction Operand Notation 3-4
Operators 3-5
Notation Conventions 3-8
Instruction Formats 3-8
Memory Instruction Format 3-8
Memory Format Instructions with a Function Code 3-9
Memory Format Jump Instructions 3-9
Branch Instruction Format 3-9
Operate Instruction Format 3-10
Floating-Point Operate Instruction Format 3-11
Floating-Point Convert Instructions 3-12
PALcode Instruction Format 3-12

Chapter 4 • Instruction Descriptions

Instruction Set Overview 4-1
Subsetting Rules 4-2
Floating-Point Subsets 4-2
Software Emulation Rules 4-2
Opcode Qualifiers 4-3
Memory Integer Load/Store Instructions 4-4
Load Address 4-5
Load Memory Data into Integer Register 4-6
Load Unaligned Memory Data into Integer Register 4-7
Load Memory Data into Integer Register Locked 4-8
Store Integer Register Data into Memory Conditional 4-11
Store Integer Register Data into Memory 4-13
Store Unaligned Integer Register Data into Memory 4-14
Control Instructions 4-15
Conditional Branch 4-17
Unconditional Branch 4-18
Jumps 4-19

Integer Arithmetic Instructions 4-21
Longword Add 4-22
Scaled Longword Add 4-23
Quadword Add 4-24
Scaled Quadword Add 4-25
Integer Signed Compare 4-26
Integer Unsigned Compare 4-27
Longword Multiply 4-28
Quadword Multiply 4-29
Unsigned Quadword Multiply High 4-30
Longword Subtract 4-31
Scaled Longword Subtract 4-32
Quadword Subtract 4-33
Scaled Quadword Subtract 4-34
Logical and Shift Instructions 4-35
Logical Functions 4-36
Conditional Move Integer 4-37
Shift Logical 4-39
Shift Arithmetic 4-40
Byte-Manipulation Instructions 4-41
Compare Byte 4-42
Extract Byte 4-44
Byte Insert 4-47
Byte Mask 4-49
Zero Bytes 4-52
Floating-Point Instructions 4-53
Floating Subsets and Floating Faults 4-53
Definitions 4-54
Encodings 4-55
Floating-Point Rounding Modes 4-55
Floating-Point Trapping Modes 4-57
Imprecise /Software Completion Trap Modes 4-58
Invalid Operation Arithmetic Trap 4-59
Division by Zero Arithmetic Trap 4-60
Overflow Arithmetic Trap 4-60
Underflow Arithmetic Trap 4-60
Inexact Result Arithmetic Trap 4-60
Integer Overflow Arithmetic Trap 4-60
Floating-Point Single-Precision Operations 4-61
FPCR Register and Dynamic Rounding Mode 4-61
Accessing the FPCR 4-63
Default Values of the FPCR 4-63
Saving and Restoring the FPCR 4-64
IEEE Standard 4-64

Memory Format Floating-Point Instructions 4-65
Load F_floating 4-66
Load G_floating 4-67
Load S_floating 4-68
Load T_floating 4-69
Store F_floating 4-70
Store G_floating 4-71
Store S_floating 4-72
Store T_floating 4-73
Branch Format Floating-Point Instructions 4-74
Conditional Branch 4-75
Floating-Point Operate Format Instructions 4-76
Copy Sign 4-78
Convert Integer to Integer 4-79
Floating-Point Conditional Move 4-80
Move from/to Floating-Point Control Register 4-82
VAX Floating Add 4-83
IEEE Floating Add 4-84
VAX Floating Compare 4-85
IEEE Floating Compare 4-86
Convert VAX Floating to Integer 4-87
Convert Integer to VAX Floating 4-88
Convert VAX Floating to VAX Floating 4-89
Convert IEEE Floating to Integer 4-90
Convert Integer to IEEE Floating 4-91
Convert IEEE Floating to IEEE Floating 4-92
VAX Floating Divide 4-93
IEEE Floating Divide 4-94
VAX Floating Multiply 4-95
IEEE Floating Multiply 4-96
VAX Floating Subtract 4-97
IEEE Floating Subtract 4-98
Miscellaneous Instructions 4-99
Call Privileged Architecture Library 4-100
Prefetch Data 4-101
Memory Barrier 4-103
Read Process Cycle Counter 4-104
Trap Barrier 4-105
VAX Compatibility Instructions 4-106
VAX Compatibility Instructions 4-107

Chapter 5 • System Architecture and Programming Implications

Introduction 5-1
Physical Memory Behavior 5-1
  Coherency of Memory Access 5-1
  Granularity of Memory Access 5-2
  Width of Memory Access 5-2
  Memory-Like Behavior 5-3
Translation Buffers and Virtual Caches 5-3
Caches and Write Buffers 5-4
Data Sharing 5-5
  Atomic Change of a Single Datum 5-5
  Atomic Update of a Single Datum 5-5
  Atomic Update of Data Structures 5-6
  Ordering Considerations for Shared Data Structures 5-7
Read/Write Ordering 5-8
  Alpha Shared Memory Model 5-9
    Architectural Definition of Processor Issue Sequence 5-10
    Definition of Processor Issue Order 5-10
    Definition of Memory Access Sequence 5-11
    Definition of Location Access Order 5-11
    Definition of Storage 5-11
    Relationship Between Issue Order and Access Order 5-12
    Definition of Before 5-12
    Definition of After 5-12
    Timeliness 5-12
  Litmus Tests 5-12
    Litmus Test 1 (Impossible Sequence) 5-12
    Litmus Test 2 (Impossible Sequence) 5-13
    Litmus Test 3 (Impossible Sequence) 5-13
    Litmus Test 4 (Sequence Okay) 5-13
    Litmus Test 5 (Sequence Okay) 5-13
    Litmus Test 6 (Sequence Okay) 5-14
    Litmus Test 7 (Impossible Sequence) 5-14
    Litmus Test 8 (Impossible Sequence) 5-14
    Litmus Test 9 (Impossible Sequence) 5-15
  Implied Barriers 5-15
  Implications for Software 5-15
    Single-Processor Data Stream 5-15
    Single-Processor Instruction Stream 5-16
    Multiple-Processor Data Stream (Including Single Processor with DMA I/O) 5-16
    Multiple-Processor Instruction Stream (Including Single Processor with DMA I/O) 5-16
    Multiple-Processor Context Switch 5-17
    Multiple-Processor Send/Receive Interrupt 5-19
  Implications for Hardware 5-19
Arithmetic Traps 5-20

Chapter 6 • Common PALcode Architecture

PALcode 6-1
PALcode Environment 6-1
Special Functions Required for PALcode 6-2
PALcode Effects on System Code 6-2
PALcode Replacement 6-2
Required PALcode Instructions 6-3
Halt 6-4
Instruction Memory Barrier 6-5

Chapter 7 • Console Subsystem Overview

Chapter 8 • Alpha VMS

Unprivileged VMS PALcode Instructions 8-1
Privileged VMS Palcode Instructions 8-8

Chapter 9 • Alpha OSF/1

Unprivileged OSF/1 PALcode Instructions 9-1
Privileged OSF/1 PALcode Instructions 9-2

Appendix A • Software Considerations

Hardware-Software Compact A-1
Instruction-Stream Considerations A-2
Instruction Alignment A-2
Multiple Instruction Issue—Factor of 3 ..... A-2
Branch Prediction and Minimizing Branch-Taken—Factor of 3 ..... A-3
Improving I-Stream Density—Factor of 3 ..... A-4
Instruction Scheduling—Factor of 3 ..... A-5
Data-Stream Considerations A-6
Data Alignment—Factor of 10 ..... A-6
Shared Data in Multiple Processors—Factor of 3 ..... A-7
Avoiding Cache/TB Conflicts—Factor of 1 ..... A-8
Sequential Read/Write—Factor of 1 ..... A-9
Prefetching—Factor of 3 ..... A-10
Code Sequences A-11
Aligned Byte/Word Memory Accesses A-11
Division A-12
Stylized Code Forms A-12
NOP A-12
Clear a Register A-13
Load Literal A-13
Register-to-Register Move A-14
Negate A-14
NOT A-14
Booleans A-14

Trap Barrier A-14
Pseudo-Operations (Stylized Code Forms) A-15
Timing Considerations: Atomic Sequences A-17

Appendix B • IEEE Floating-Point Conformance

Alpha Choices for IEEE Options B-1
Alpha Hardware Support of Software Exception Handlers B-2
Mapping to IEEE Standard B-3

Appendix C • Instruction Encodings

Memory Format Instructions C-1
Branch Format Instructions C-2
Operate Format Instructions C-2
Floating-Point Operate Format C-3
IEEE Floating-Point Instructions C-3
VAX Floating-Point Instructions C-5
Required PALcode Function Codes C-5
Opcodes Reserved to PALcode C-6
Opcodes Reserved to Digital C-6
Opcode Summary C-6

Index

Figures

1-1 Instruction Format Overview 1-4
2-1 Byte Format 2-1
2-2 Word Format 2-1
2-3 Longword Format 2-2
2-4 Quadword Format 2-2
2-5 F_floating Datum 2-3
2-6 F_floating Register Format 2-3
2-7 G_floating Datum 2-4
2-8 G_floating Format 2-4
2-9 D_floating Datum 2-5
2-10 D_floating Register Format 2-5
2-11 S_floating Datum 2-7
2-12 S_floating Register Format 2-7
2-13 T_floating Datum 2-8
2-14 T_floating Register Format 2-9
2-15 Longword Integer Datum 2-9
2-16 Longword Integer Floating-Register Format 2-10
2-17 Quadword Integer Datum 2-10
2-18 Quadword Integer Floating-Register Format 2-10
3-1 Memory Instruction Format 3-8
3-2 Memory Instruction with Function Code Format 3-9
3-3 Branch Instruction Format 3-9
3-4 Operate Instruction Format 3-10
3-5 Floating-Point Operate Instruction Format 3-11

3-6 PALcode Instruction Format 3-12
4-1 Floating-Point Control Register (FPCR) Format 4-61
B-1 IEEE Trap Handling Behavior B-3

Tables

2-1 F_floating Load Exponent Mapping 2-3
2-2 S_floating Load Exponent Mapping 2-7
3-1 Operand Notation 3-3
3-2 Operand Value Notation 3-3
3-3 Expression Operand Notation 3-3
3-4 Operators 3-5
4-1 Opcode Qualifiers 4-3
4-2 Memory Integer Load/Store Instructions 4-4
4-3 Control Instructions Summary 4-16
4-4 Jump Instructions Branch Prediction 4-20
4-5 Integer Arithmetic Instructions Summary 4-21
4-6 Logical and Shift Instructions Summary 4-35
4-7 Byte-Manipulation Instructions Summary 4-41
4-8 Floating-Point Control Register (FPCR) Bit Descriptions 4-62
4-9 Memory Format Floating-Point Instructions Summary 4-65
4-10 Floating-Point Branch Instructions Summary 4-74
4-11 Floating-Point Operate Instructions Summary 4-76
4-12 Miscellaneous Instructions Summary 4-99
4-13 VAX Compatibility Instructions Summary 4-106
5-1 Processor Issue Order 5-10
5-2 Location Access Order 5-11
6-1 Required PALcode Instructions 6-3
8-1 Unprivileged VMS PALcode Instruction Summary 8-1
8-2 Privileged VMS PALcode Instructions Summary 8-8
9-1 Unprivileged OSF/1 PALcode Instruction Summary 9-1
9-2 Privileged OSF/1 PALcode Instruction Summary 9-2
A-1 Decodable Pseudo-Operations (Stylized Code Forms) A-15
B-1 IEEE Floating-Point Trap Handling B-4
B-2 IEEE Standard Charts B-9
C-1 Memory Format Instruction Opcodes C-1
C-2 Memory Format Instructions with a Function Code C-1
C-3 Memory Format Branch Instruction Opcodes C-2
C-4 Branch Format instruction Opcodes C-2
C-5 Operate Format Instruction Opcodes and Function Codes C-2
C-6 Function Codes for Floating Data Type Independent Operations C-3
C-7 IEEE Floating-Point Instruction Function Codes C-3
C-8 VAX Floating-Point Instruction Function Codes C-5
C-9 Required PALcode Function Codes C-5
C-10 Opcodes Reserved for PALcode C-6
C-11 Opcodes Reserved for Digital C-6
C-12 Opcode Summary C-7
C-13 Key to Opcode Summary (Table C-12) C-7

Preface

This book describes Digital's next generation RISC architecture. It is directly derived from sections of the *Alpha System Reference Manual* and is an accurate representation of the described parts of the Alpha architecture.

Chapter 1 • Introduction

Alpha is a 64-bit load/store RISC architecture that is designed with particular emphasis on the three elements that most affect performance: clock speed, multiple instruction issue, and multiple processors.

The Alpha architects examined and analyzed current and theoretical RISC architecture design elements and developed high-performance alternatives for the Alpha architecture. The architects adopted only those design elements that appeared valuable for a projected 25-year design horizon. Thus, Alpha becomes the first 21st century computer architecture.

The Alpha architecture is designed to avoid bias toward any particular operating system or programming language. Alpha initially supports the VAX VMS and OSF/1 (UNIX) operating systems, and supports simple software migration from applications that run on those operating systems.

This handbook describes in detail how Alpha is designed to be the leadership 64-bit architecture of the computer industry.

• The Alpha Approach to RISC Architecture

Alpha Is a True 64-Bit Architecture

Alpha was designed as a 64-bit architecture. All registers are 64 bits in length and all operations are performed between 64-bit registers. It is not a 32-bit architecture that was later expanded to 64 bits.

Alpha Is Designed for Very High-Speed Implementations

The instructions are very simple. All instructions are 32 bits in length. Memory operations are either loads or stores. All data manipulation is done between registers.

The Alpha architecture facilitates pipelining multiple instances of the same operations because there are no special registers and no condition codes.

The instructions interact with each other only by one instruction writing a register or memory and another instruction reading from the same place. That makes it particularly easy to build implementations that issue multiple instructions every CPU cycle. (The first implementation issues two instructions per cycle.)

Alpha makes it easy to maintain binary compatibility across multiple implementations and easy to maintain full speed on multiple-issue implementations. For example, there are no implementation-specific pipeline timing hazards, no load-delay slots, and no branch-delay slots.

Alpha's Approach to Byte Manipulation

The Alpha architecture does byte shifting and masking with normal 64-bit register-to-register instructions, crafted to keep instruction sequences short.

Alpha does not include single-byte store instructions. This has several advantages:

- Cache and memory implementations need not include byte shift-and-mask logic, and sequencer logic need not perform read-modify-write on memory locations. Such logic is awkward for high-speed implementation and tends to slow down cache access to normal 32-bit or 64-bit aligned quantities.
- Alpha's approach to byte manipulation makes it easier to build a high-speed error-correcting write-back cache, which is often needed to keep a very fast RISC implementation busy.
- Alpha's approach can make it easier to pipeline multiple byte operations.

Alpha's Approach to Arithmetic Traps

Alpha lets the software implementor determine the precision of arithmetic traps. With the Alpha architecture, arithmetic traps (such as overflow and underflow) are imprecise—they can be delivered an arbitrary number of instructions after the instruction that triggered the trap. Also, traps from many different instructions can be reported at once. That makes implementations that use pipelining and multiple issue substantially easier to build.

However, if precise arithmetic exceptions are desired, trap barrier instructions can be explicitly inserted in the program to force traps to be delivered at specific points.

Alpha's Approach to Multiprocessor Shared Memory

As viewed from a second processor (including an I/O device), a sequence of reads and writes issued by one processor may be arbitrarily reordered by an implementation. This allows implementations to use multibank caches, bypassed write buffers, write merging, pipelined writes with retry on error, and so forth. If strict ordering between two accesses must be maintained, explicit memory barrier instructions can be inserted in the program.

The basic multiprocessor interlocking primitive is a RISC-style load_locked, modify, store_conditional sequence. If the sequence runs without interrupt, exception, an interfering write from another processor, or a CALL_PAL instruction, then the conditional store succeeds. Otherwise, the store fails and the program eventually must branch back and retry the sequence. This style of interlocking scales well with very fast caches, and makes Alpha an especially attractive architecture for building multiple-processor systems.

Alpha Instructions Include Hints for Achieving Higher Speed

A number of Alpha instructions include hints for implementations, all aimed at achieving higher speed.

- Calculated jump instructions have a target hint that can allow much faster subroutine calls and returns.
- There are prefetching hints for the memory system that can allow much higher cache hit rates.
- There are granularity hints for the virtual-address mapping that can allow much more effective use of translation lookaside buffers for large contiguous structures.

PALcode—Alpha's Very Flexible Privileged Software Library

A Privileged Architecture Library (PALcode) is a set of subroutines that are specific to a particular Alpha operating system implementation. These subroutines provide operating-system primitives for context switching, interrupts, exceptions, and memory management. PALcode is similar to the BIOS libraries that are provided in personal computers

PALcode subroutines are invoked by implementation hardware or by software CALL_PAL instructions.

PALcode is written in standard machine code with some implementation-specific extensions to provide access to low-level hardware.

One version of PALcode lets Alpha implementations run the full VMS operating system by mirroring many of the VAX VMS features. The VMS PALcode instructions let Alpha run VMS with little more hardware than that found on a conventional RISC machine: the PAL mode bit itself, plus 4 extra protection bits in each Translation Buffer entry.

Another version of PALcode lets Alpha implementations run the OSF/1 operating system by mirroring many of the RISC ULTRIX features. Other versions of PALcode can be developed for real-time, teaching, and other applications.

PALcode makes Alpha an especially attractive architecture for multiple operating systems.

Alpha and Programming Languages

Alpha is an attractive architecture for compiling a large variety of programming languages. Alpha has been carefully designed to avoid bias toward one or two programming languages. For example:

- Alpha does not contain a subroutine call instruction that moves a register window by a fixed amount. Thus, Alpha is a good match for programming languages with many parameters and programming languages with no parameters.
- Alpha does not contain a global integer overflow enable bit. Such a bit would need to be changed at every subroutine boundary when a FORTRAN program calls a C program.

Data Format Overview

Alpha is a load/store RISC architecture with the following data characteristics:

- All operations are done between 64-bit registers.
- Memory is accessed via 64-bit virtual little-endian byte addresses.
- There are 32 integer registers and 32 floating-point registers.
- Longword (32-bit) and quadword (64-bit) integers are supported.
- Four floating-point data types are supported:
  - VAX F_floating (32-bit)
  - VAX G_floating (64-bit)
  - IEEE single (32-bit)
  - IEEE double (64-bit)

• Instruction Format Overview

As shown in Figure 1-1, Alpha instructions are all 32 bits in length. As represented in Figure 1-1, there are four major instruction format classes that contain 0, 1, 2, or 3 register fields. All formats have a 6-bit opcode.

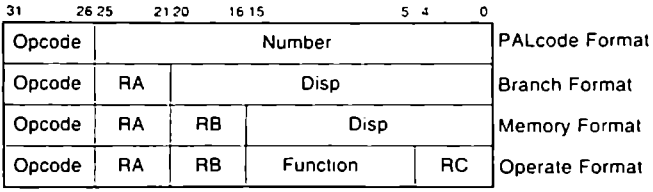


Figure 1-1 • Instruction Format Overview

- **PALcode instructions** specify, in the function code field, one of a few dozen complex operations to be performed.
- **Conditional branch instructions** test register Ra and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in register Ra.
- **Load and store instructions** move longwords or quadwords between register Ra and memory, using Ra plus a signed 16-bit displacement as the memory address.
- **Operate instructions** for floating-point and integer operations are both represented in Figure 1-1 by the operate format illustration and are as follows:
  - Floating-point operations use Ra and Rb as source registers, and write the result in register Rc. There is an 11-bit extended opcode in the function field.
  - Integer operations use Ra and Rb or an 8-bit literal as the source operand, and write the result in register Rc.

    Integer operate instructions can use the Rb field and part of the function field to specify an 8-bit literal. There is a 7-bit extended opcode in the function field.

• Instruction Overview

PALcode Instructions

As described above, a Privileged Architecture Library (PALcode) is a set of subroutines that is specific to a particular Alpha operating-system implementation. These subroutines can be invoked by hardware or by software CALL_PAL instructions, which use the function field to vector to the specified subroutine.

Branch Instructions

Conditional branch instructions can test a register for positive/negative or for zero/nonzero. They can also test integer registers for even/odd.

Unconditional branch instructions can write a return address into a register.

There is also a calculated jump instruction that branches to an arbitrary 64-bit address in a register.

Load/Store Instructions

Load and store instructions move either 32-bit or 64-bit aligned quantities from and to memory. Memory addresses are flat 64-bit virtual addresses, with no segmentation.

The VAX floating-point load/store instructions swap words to give a consistent register format for floating-point operations.

A 32-bit integer datum is placed in a register in a canonical form that makes 33 copies of the high bit of the datum. A 32-bit floating-point datum is placed in a register in a canonical form that extends the exponent by 3 bits and extends the fraction with 29 low-order zeros. The 32-bit operates preserve these canonical forms.

There are facilities for doing byte manipulation in registers, eliminating the need for 8-bit or 16-bit load/store instructions.

Compilers, as directed by user declarations, can generate any mixture of 32-bit and 64-bit operations. The Alpha architecture has no 32/64 mode bit.

Integer Operate Instructions

The integer operate instructions manipulate full 64-bit values, and include the usual assortment of arithmetic, compare, logical, and shift instructions.

There are just three 32-bit integer operates: add, subtract, and multiply. They differ from their 64-bit counterparts only in overflow detection and in producing 32-bit canonical results.

There is no integer divide instruction.

The Alpha architecture also supports the following additional operations:

- Scaled add/subtract instructions for quick subscript calculation
- 128-bit multiply for division by a constant, and multiprecision arithmetic
- Conditional move instructions for avoiding branch instructions
- An extensive set of in-register byte and word manipulation instructions

Integer overflow trap enable is encoded in the function field of each instruction, rather than kept in a global state bit. Thus, for example, both ADDQ/V and ADDQ opcodes exist for specifying 64-bit ADD with and without overflow checking. That makes it easier to pipeline implementations.

Floating-Point Operate Instructions

The floating-point operate instructions include four complete sets of VAX and IEEE arithmetic instructions, plus instructions for performing conversions between floating-point and integer quantities.

In addition to the operations found in conventional RISC architectures, Alpha includes conditional move instructions for avoiding branches and merge sign/exponent instructions for simple field manipulation.

The arithmetic trap enables and rounding mode are encoded in the function field of each instruction, rather then kept in global state bits. That makes it easier to pipeline implementations.

Instruction Set Characteristics

Alpha instruction set characteristics are as follows:

- All instructions are 32 bits long and have a regular format.
- There are 32 integer registers (R0 through R31), each 64 bits wide. R31 reads as zero, and writes to R31 are ignored.
- There are 32 floating-point registers (F0 through F31), each 64 bits wide. F31 reads as zero, and writes to F31 are ignored.
- All integer data manipulation is between integer registers, with up to two variable register source operands (one may be an 8-bit literal), and one register destination operand.
- All floating-point data manipulation is between floating-point registers, with up to two register source operands and one register destination operand.
- All memory reference instructions are of the load/store type that move data between registers and memory.
- There are no branch condition codes. Branch instructions test an integer or floating-point register value, which may be the result of a previous compare.
- Integer and logical instructions operate on quadwords.
- Floating-point instructions operate on G_floating, F_floating, IEEE double, and IEEE single operands. D_floating "format compatibility," in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, is also provided.
- A minimal number of VAX compatibility instructions are included.

Terminology and Conventions

The following sections describe the terminology and conventions used in this book.

Numbering

All numbers are decimal unless otherwise indicated. Where there is ambiguity, numbers other than decimal are indicated with the name of the base in subscript form, for example, 10_{16}.

Security Holes

A security hole is an error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed.

Security holes exist when unprivileged software (that is, software running outside of kernel mode) can:

- Affect the operation of another process without authorization from the operating system;
- Amplify its privilege without authorization from the operating system; or
- Communicate with another process, either overtly or covertly, without authorization from the operating system.

The Alpha architecture has been designed to contain no architectural security holes. Hardware (processors, buses, controllers, and so on) and software should likewise be designed to avoid security holes.

UNPREDICTABLE and UNDEFINED

In this book, the terms UNPREDICTABLE and UNDEFINED are used. Their meanings are quite different and must be carefully distinguished. One key difference is that only privileged software (that is, software running in kernel mode) may trigger UNDEFINED operations, whereas either privileged or unprivileged software may trigger UNPREDICTABLE results or occurrences. A second key difference is that UNPREDICTABLE results and occurrences do not disrupt the basic operation of the processor; the processor continues to execute instructions in its normal manner. In contrast, UNDEFINED operation may halt the processor or cause it to lose information.

A result specified as UNPREDICTABLE may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values. Operations that produce UNPREDICTABLE results may also produce exceptions.

UNPREDICTABLE results must not be security holes.

Specifically, UNPREDICTABLE results must not:

- Depend upon, or be a function of, the contents of memory locations or registers that are inaccessible to the current process in the current access mode.

Also, operations that may produce UNPREDICTABLE results must not:

- Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or
- Halt or hang the system or any of its components.

For example, a security hole would exist if some UNPREDICTABLE result depended on the value of a register in another process, on the contents of processor temporary registers left behind by some previously running process, or on a sequence of actions of different processes.

An occurrence specified as UNPREDICTABLE may happen or not based on an arbitrary choice function. The choice function is subject to the same constraints as are UNPREDICTABLE results and, in particular, must not constitute a security hole.

Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.

Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation. UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions. Only privileged software (that is, software running in kernel mode) may trigger UNDEFINED operations.

Ranges and Extents

Ranges are specified by a pair of numbers separated by a ".." and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive. For example, bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

ALIGNED and UNALIGNED

In this document the terms ALIGNED and NATURALLY ALIGNED are used interchangeably to refer to data objects that are powers of two in size. An aligned datum of size 2**N is stored in memory at a byte address that is a multiple of 2**N, that is, one that has N low-order zeros. Thus, an aligned 64-byte stack frame has a memory address that is a multiple of 64.

If a datum of size 2**N is stored at a byte address that is not a multiple of 2**N, it is called UNALIGNED.

Must Be Zero (MBZ)

Fields specified as Must be Zero (MBZ) must never be filled by software with a non-zero value. These fields may be used at some future time. If the processor encounters a non-zero value in a field specified as MBZ, an Illegal Operand exception occurs.

Read As Zero (RAZ)

Fields specified as Read as Zero (RAZ) return a zero when read.

Should Be Zero (SBZ)

Fields specified as Should be Zero (SBZ) should be filled by software with a zero value. Non-zero values in SBZ fields produce UNPREDICTABLE results and may produce extraneous instruction-issue delays.

Ignore (IGN)

Fields specified as Ignore (IGN) are ignored when written.

Implementation Dependent (IMP)

Fields specified as Implementation Dependent (IMP) may be used for implementation-specific purposes. Each implementation must document fully the behavior of all fields marked as IMP by the Alpha specification.

Figure Drawing Conventions

Figures that depict registers or memory follow the convention that increasing addresses run right to left and top to bottom.

Macro Code Example Conventions

All instructions in macro code examples are either listed in Chapter 4 or are stylized code forms found in Appendix A.

Chapter 2 ▪ Basic Architecture

▪ Addressing

The basic addressable unit in Alpha is the 8-bit byte. Virtual addresses are 64 bits long. An implementation may support a smaller virtual address space. The minimum virtual address size is 43 bits.

Virtual addresses as seen by the program are translated into physical memory addresses by the memory management mechanism.

▪ Data Types

Following are descriptions of the Alpha architecture data types.

Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left, 0 through 7, as shown in Figure 2-1.

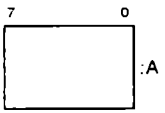


Figure 2-1 ▪ Byte Format

A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha by the extract, mask, insert, and zap instructions.

Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 15, as shown in Figure 2-2.

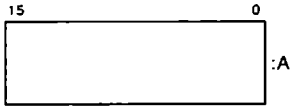


Figure 2-2 ▪ Word Format

A word is specified by its address, the address of the byte containing bit 0.

A word is a 16-bit value. The word is only supported in Alpha by the extract, mask, and insert instructions.

Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 31, as shown in Figure 2-3.

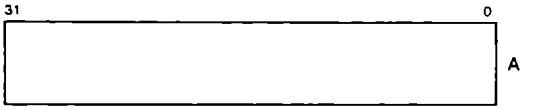


Figure 2-3 • Longword Format

A longword is specified by its address A, the address of the byte containing bit 0. A longword is a 32-bit value.

When interpreted arithmetically, a longword is a two's-complement integer with bits of increasing significance from 0 through 30. Bit 31 is the sign bit. The longword is only supported in Alpha by sign-extended load and store instructions and by longword arithmetic instructions.

Note

Alpha implementations will impose a significant performance penalty when accessing longword operands that are not naturally aligned. (A naturally aligned longword has zero as the low-order two bits of its address.)

Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 63, as shown in Figure 2-4.

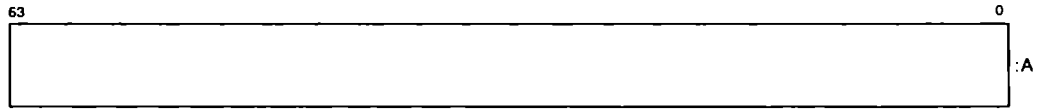


Figure 2-4 • Quadword Format

A quadword is specified by its address A, the address of the byte containing bit 0. A quadword is a 64-bit value. When interpreted arithmetically, a quadword is either a two's-complement integer with bits of increasing significance from 0 through 62 and bit 63 as the sign bit, or an unsigned integer with bits of increasing significance from 0 through 63.

Note

Alpha implementations will impose a significant performance penalty when accessing quadword operands that are not naturally aligned. (A naturally aligned quadword has zero as the low-order three bits of its address.)

VAX Floating-Point Formats

VAX floating-point numbers are stored in one set of formats in memory and in a second set of formats in registers. The floating-point load and store instructions convert between these formats purely by rearranging bits; no rounding or range-checking is done by the load and store instructions.

F_floating

An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2-5.

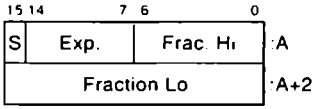


Figure 2-5 • F_floating Datum

An F_floating operand occupies 64 bits in a floating register, left-justified in the 64-bit register, as shown in Figure 2-6.

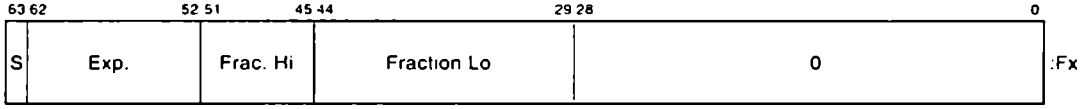


Figure 2-6 • F_floating Register Format

The F_floating load instruction reorders bits on the way in from memory, expands the exponent from 8 to 11 bits, and sets the low-order fraction bits to zero. This produces in the register an equivalent G_floating number suitable for either F_floating or G_floating operations. The mapping from 8-bit memory-format exponents to 11-bit register-format exponents is shown in Table 2-1.

Table 2-1 • F_floating Load Exponent Mapping

Memory <14:7>	Register <62:52>	
1 1111111	1 000 1111111	
1 xxxxxxx	1 000 xxxxxxx	(xxxxxxx not all 1's)
0 xxxxxxx	0 111 xxxxxxx	(xxxxxxx not all 0's)
0 0000000	0 000 0000000	

This mapping preserves both normal values and exceptional values.

The F_floating store instruction reorders register bits on the way to memory and does no checking of the low-order fraction bits. Register bits <61:59> and <28:0> are ignored by the store instruction.

An F_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits <14:7> an excess-128 binary exponent, and bits <6:0> and <31:16> a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0.

If the result of a VAX floating-point format instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..255 indicate true binary exponents of −127..127. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take an arithmetic exception. The value of an F_floating datum is in the approximate range 0.29*10**−38..1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, typically 7 decimal digits.

Note

Alpha implementations will impose a significant performance penalty when accessing F_floating operands that are not naturally aligned. (A naturally aligned F_floating datum has zero as the low-order two bits of its address.)

G_floating

A G_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2-7.

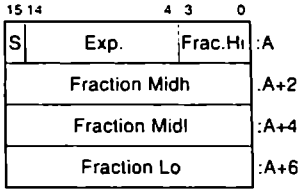


Figure 2-7 • G_floating Datum

A G_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2-8.

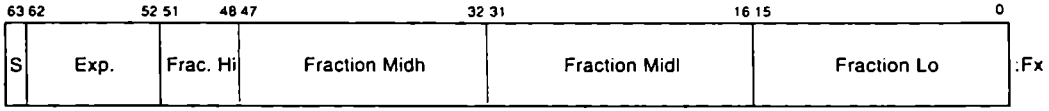


Figure 2-8 • G_floating Format

A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits <14:4> an excess-1024 binary exponent, and bits <3:0> and <63:16> a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are from 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0.

If the result of a floating-point instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..2047 indicate true binary exponents of -1023..1023. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take a user-visible arithmetic exception. The value of a G_floating datum is in the approximate range 0.56*10**-308..0.9*10**308. The precision of a G_floating datum is approximately one part in 2**52, typically 15 decimal digits.

Note

Alpha implementations will impose a significant performance penalty when accessing G_floating operands that are not naturally aligned. (A naturally aligned G_floating datum has zero as the low-order three bits of its address.)

D_floating

A D_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2-9.

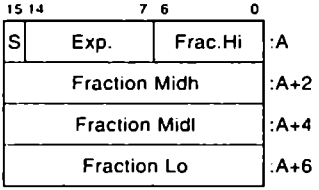


Figure 2-9 • D_floating Datum

A D_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2-10.

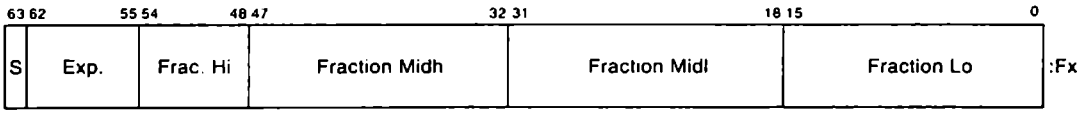


Figure 2-10 • D_floating Register Format

The reordering of bits required for a D_floating load or store are identical to those required for a G_floating load or store. The G_floating load and store instructions are therefore used for loading or storing D_floating data.

A D_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of a D_floating datum is identical to an F_floating datum except for 32 additional low significance fraction bits. Within the fraction, bits of increasing significance are from 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions and approximate range of values is the same for D_floating as F_floating. The precision of a D_floating datum is approximately one part in 2**55, typically 16 decimal digits.

Note

D_floating is not a fully supported data type: no D_floating arithmetic operations are provided in the architecture. For backward compatibility, exact D_floating arithmetic may be provided via software emulation. D_floating "format compatibility" in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, can be obtained via conversions to G_floating, G arithmetic operations, then conversion back to D_floating.

Note

Alpha implementations will impose a significant performance penalty on access to D_floating operands that are not naturally aligned. (A naturally aligned D_floating datum has zero as the low-order three bits of its address.)

IEEE Floating-Point Formats

The IEEE standard for binary floating-point arithmetic, ANSI/IEEE 754-1985, defines four floating-point formats in two groups, basic and extended, each having two widths, single and double. The Alpha architecture supports the basic single and double formats, with the basic double format serving as the extended single format. The values representable within a format are specified by using three integer parameters:

1. P—the number of fraction bits
2. Emax—the maximum exponent
3. Emin—the minimum exponent

Within each format, only the following entities are permitted:

1. Numbers of the form (−1)**S × 2**E × b(0).b(1)b(2)..b(P−1) where:
   a. S = 0 or 1
   b. E = any integer between Emin and Emax, inclusive
   c. b(n) = 0 or 1
2. Two infinities—positive and negative
3. At least one Signaling NaN
4. At least one Quiet NaN

NaN is an acronym for Not-a-Number. A NaN is an IEEE floating-point bit pattern that represents something other than a number. NaNs come in two forms: Signaling NaNs and Quiet NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic

enhancements. Quiet NaNs provide retrospective diagnostic information regarding previous invalid or unavailable data and results. Signaling NaNs signal an invalid operation when they are an operand to an arithmetic instruction, and may generate an arithmetic exception. Quiet NaNs propagate through almost every operation without generating an arithmetic exception.

Arithmetic with the infinities is handled as if the operands were of arbitrarily large magnitude. Negative infinity is less than every finite number; positive infinity is greater than every finite number.

S_Floating

An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2-11.

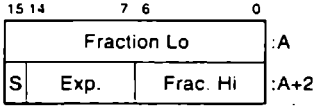


Figure 2-11 • S_floating Datum

An S_floating operand occupies 64 bits in a floating register, left-justified in the 64-bit register, as shown in Figure 2-12.

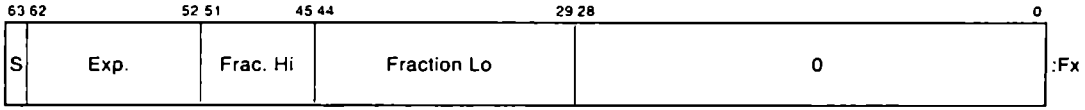


Figure 2-12 • S_floating Register Format

The S_floating load instruction reorders bits on the way in from memory, expanding the exponent from 8 to 11 bits, and sets the low-order fraction bits to zero. This produces in the register an equivalent T_floating number, suitable for either S_floating or T_floating operations. The mapping from 8-bit memory-format exponents to 11-bit register-format exponents is shown in Table 2-2.

Table 2-2 • S_floating Load Exponent Mapping

Memory <30:23>	Register <62:52>	
1 1111111	1 111 1111111	
1 xxxxxxx	1 000 xxxxxxx	(xxxxxxx not all 1's)
0 xxxxxxx	0 111 xxxxxxx	(xxxxxxx not all 0's)
0 0000000	0 000 0000000	

This mapping preserves both normal values and exceptional values. Note that the mapping for all 1's differs from that of F_floating load, since for S_floating all 1's is an exceptional value and for F_floating all 1's is a normal value.

The S_floating store instruction reorders register bits on the way to memory and does no checking of the low-order fraction bits. Register bits <61:59> and <28:0> are ignored by the store instruction. The S_floating load instruction does no checking of the input.

The S_floating store instruction does no checking of the data; the preceding operation should have specified an S_floating result.

An S_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of an S_floating datum is sign magnitude with bit 31 the sign bit, bits <30:23> an excess-127 binary exponent, and bits <22:0> a 23-bit fraction.

The value (V) of an S_floating number is inferred from its constituent sign (S), exponent (E), and fraction (F) fields as follows:

1. If E=255 and F<>0, then V is NaN, regardless of S.
2. If E=255 and F=0, then V = (−1)**S × Infinity.
3. If 0 < E < 255, then V = (−1)**S × 2**(E−127) × (1.F).
4. If E=0 and F<>0, then V = (−1)**S × 2**(−126) × (0.F).
5. If E=0 and F=0, then V = (−1)**S × 0 (zero).

Floating-point operations on S_floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.

Note

Alpha implementations will impose a significant performance penalty when accessing S_floating operands that are not naturally aligned. (A naturally aligned S_floating datum has zero as the low-order two bits of its address.)

T_floating

An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2-13.

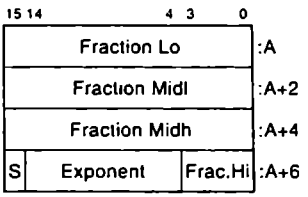


Figure 2-13 • T_floating Datum

A T_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2-14.

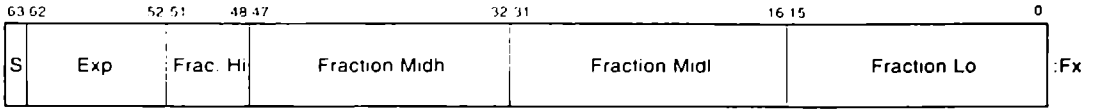


Figure 2-14 • T_floating Register Format

The T_floating load instruction performs no bit reordering on input, nor does it perform checking of the input data.

The T_floating store instruction performs no bit reordering on output. This instruction does no checking of the data; the preceding operation should have specified a T_floating result.

A T_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a T_floating datum is sign magnitude with bit 63 the sign bit, bits <62:52> an excess-1023 binary exponent, and bits <51:0> a 52-bit fraction.

The value (V) of a T_floating number is inferred from its constituent sign (S), exponent (E), and fraction (F) fields as follows:

1. If E=2047 and F<>0, then V is NaN, regardless of S.
2. If E=2047 and F=0, then V = (−1)**S × Infinity.
3. If 0 < E < 2047, then V = (−1)**S × 2**(E−1023) × (1.F).
4. If E=0 and F<>0, then V = (−1)**S × 2**(−1022) × (0.F).
5. If E=0 and F=0, then V = (−1)**S × 0 (zero).

Floating-point operations on T_floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.

Note

Alpha implementations will impose a significant performance penalty when accessing T_floating operands that are not naturally aligned. (A naturally aligned T_floating datum has zero as the low-order three bits of its address.)

Longword Integer Format in Floating-Point Unit

A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2-15.

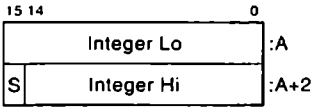


Figure 2-15 • Longword Integer Datum

A longword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2-16.

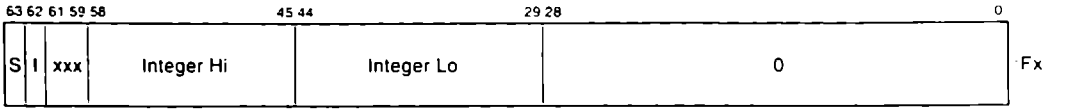


Figure 2-16 • Longword Integer Floating-Register Format

There is no explicit longword load or store instruction; the S_floating load/store instructions are used to move longword data into or out of the floating registers. The register bits <61:59> are set by the S_floating load exponent mapping. They are ignored by S_floating store. They are also ignored in operands of a longword integer operate instruction, and they are set to 000 in the result of a longword operate instruction.

The register format bit <62>, "I", in Figure 2-16 is part of the Integer Hi field in Figure 2-15 and represents the high-order bit of that field. Bits <58:45> of Figure 2-16 are the remaining bits of the Integer Hi field of Figure 2-15.

Note

Alpha implementations will impose a significant performance penalty when accessing longwords that are not naturally aligned. (A naturally aligned longword datum has zero as the low-order two bits of its address.)

Quadword Integer Format in Floating-Point Unit

A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2-17.

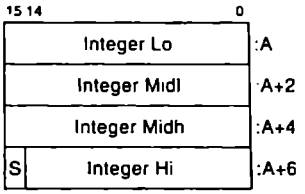


Figure 2-17 • Quadword Integer Datum

A quadword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2-18.

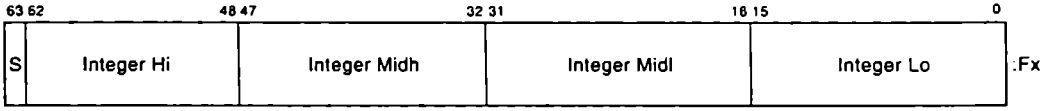


Figure 2-18 • Quadword Integer Floating-Register Format

There is no explicit quadword load or store instruction, the T_floating load/store instructions are used to move quadword data into or out of the floating registers.

The T_floating load instruction performs no bit reordering on input. The T_floating store instruction performs no bit reordering on output. This instruction does no checking of the data; when used to store quadwords, the preceding operation should have specified a quadword result.

Note

Alpha implementations will impose a significant performance penalty when accessing quadwords that are not naturally aligned. (A naturally aligned quadword datum has zero as the low-order three bits of its address.)

Data Types with No Hardware Support

The following VAX data types are not directly supported in Alpha hardware.

- Octaword
- H_floating
- D_floating (except load/store and convert to/from G_floating)
- Variable-Length Bit Field
- Character String
- Trailing Numeric String
- Leading Separate Numeric String
- Packed Decimal String

Chapter 3 • Instruction Formats

• Alpha Registers

Each Alpha processor has a set of registers that hold the current processor state. If an Alpha system contains multiple Alpha processors, there are multiple per-processor sets of these registers.

Program Counter

The Program Counter (PC) is a special register that addresses the instruction stream. As each instruction is decoded, the PC is advanced to the next sequential instruction. This is referred to as the *updated PC*. Any instruction that uses the value of the PC will use the updated PC . The PC includes only bits <63:2> with bits <1:0> treated as RAZ/IGN. This quantity is a longword-aligned byte address. The PC is an implied operand on conditional branch and subroutine jump instructions. The PC is not accessible as an integer register.

Integer Registers

There are 32 integer registers (R0 through R31), each 64 bits wide.

Register R31 is assigned special meaning by the Alpha architecture:

- When R31 is specified as a register source operand, a zero-valued operand is supplied.

For all cases except the Unconditional Branch and Jump instructions, results of an instruction that specifies R31 as a destination operand are discarded. Also, it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. It is implementation dependent to what extent the instruction is actually executed once it has been fetched. It is also UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

There are some interesting cases involving R31 as a destination:

– STx_C R31,disp(Rb)

  Although this might seem like a good way to zero out a shared location and reset the lock_flag, this instruction causes the lock_flag and virtual location (Rbv + SEXT(disp)) to become UNPREDICTABLE.

– LDx_L R31,disp(Rb)

  This instruction produces no useful result since it causes both lock_flag and locked_physical_address to become UNPREDICTABLE.

  Unconditional Branch (BR and BSR) and Jump (JMP, JSR, RET, and JSR_COROUTINE) instructions, when R31 is specified as the Ra operand, execute normally and update the PC with the target virtual address. Of course, no PC value can be saved in R31.

Floating-Point Registers

There are 32 floating-point registers (F0 through F31), each 64 bits wide.

When F31 is specified as a register source operand, a true zero-valued operand is supplied. See *Definitions* in Chapter 4 for a definition of true zero.

Results of an instruction that specifies F31 as a destination operand are discarded and it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. In this case, it is implementation-dependent to what extent the instruction is actually executed once it has been fetched. It is also UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

A floating-point instruction that operates on single-precision data reads all bits <63:0> of the source floating-point register. A floating-point instruction that produces a single-precision result writes all bits <63:0> of the destination floating-point register.

Lock Registers

There are two per-processor registers associated with the LDx_L and STx_C instructions, the lock_flag and the locked_physical_address register. The use of these registers is described in *Memory Integer Load/Store Instructions* in Chapter 4.

Optional Registers

Some Alpha implementations may include optional memory prefetch or VAX compatibility processor registers.

Memory Prefetch Registers

If the prefetch instructions FETCH and FETCH_M are implemented, an implementation will include two sets of state prefetch registers used by those instructions. The use of these registers is described in *Miscellaneous Instructions* in Chapter 4. These registers are not directly accessible by software and are listed for completeness.

VAX Compatibility Register

The VAX compatibility instructions RC and RS include the intr_flag register, as described in *VAX Compatibility Instructions* in Chapter 4.

Notation

The notation used to describe the operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax.

Operand Notation

Tables Table 3-1, 3-2, and 3-3 list the notation for the operands, the operand values, and the other expression operands.

Table 3-1 • Operand Notation

Notation	Meaning
Ra	An integer register operand in the Ra field of the instruction.
Rb	An integer register operand in the Rb field of the instruction.
#b	An integer literal operand in the Rb field of the instruction.
Rc	An integer register operand in the Rc field of the instruction.
Fa	A floating-point register operand in the Ra field of the instruction.
Fb	A floating-point register operand in the Rb field of the instruction.
Fc	A floating-point register operand in the Rc field of the instruction.

Table 3-2 • Operand Value Notation

Notation	Meaning
Rav	The value of the Ra operand. This is the contents of register Ra.
Rbv	The value of the Rb operand. This could be the contents of register Rb, or a zero-extended 8-bit literal in the case of an Operate format instruction.
Fav	The value of the floating point Fa operand. This is the contents of register Fa.
Fbv	The value of the floating point Fb operand. This is the contents of register Fb.

Table 3-3 • Expression Operand Notation

Notation	Meaning
IPR_x	Contents of Internal Processor Register x
IPR_SP[mode]	Contents of the per-mode stack pointer selected by mode
PC	Updated PC value
Rn	Contents of integer register n
Fn	Contents of floating-point register n
X[m]	Element m of array X

Instruction Operand Notation

The notation used to describe instruction operands follows from the operand specifier notation used in the *VAX Architecture Standard*. Instruction operands are described as follows:

<name>.<access type><data type>

<name>
Specifies the instruction field (Ra, Rb, Rc, or disp) and register type of the operand (integer or floating). It can be one of the following:

Name	Meaning
disp	The displacement field of the instruction.
fnc	The PAL function field of the instruction.
Ra	An integer register operand in the Ra field of the instruction.
Rb	An integer register operand in the Rb field of the instruction.
#b	An integer literal operand in the Rb field of the instruction.
Rc	An integer register operand in the Rc field of the instruction.
Fa	A floating-point register operand in the Ra field of the instruction.
Fb	A floating-point register operand in the Rb field of the instruction.
Fc	A floating-point register operand in the Rc field of the instruction.

<access type>
Is a letter denoting the operand access type:

Access Type	Meaning
a	The operand is used in an address calculation to form an effective address. The data type code that follows indicates the units of addressability (or scale factor) applied to this operand when the instruction is decoded. For example: ".al" means scale by 4 (longwords) to get byte units (used in branch displacements); ".ab" means the operand is already in byte units (used in load/store instructions).
i	The operand is an immediate literal in the instruction.
r	The operand is read only.
m	The operand is both read and written.
w	The operand is write only.

<data type>
Is a letter denoting the data type of the operand:

Data Type	Meaning
b	Byte
f	F_floating
g	G_floating
l	Longword
q	Quadword
s	IEEE single floating (S_floating)
t	IEEE double floating (T_floating)
w	Word
x	The data type is specified by the instruction

Operators

The operators shown in Table 3-4 are used:

Table 3-4 • Operators

Operator	Meaning
!	Comment delimiter
+	Addition
-	Subtraction
*	Signed multiplication
*U	Unsigned multiplication
**	Exponentiation (left argument raised to right argument)
/	Division
←	Replacement
||	Bit concatenation
()	Indicates explicit operator precedence
(x)	Contents of memory location whose address is x
x<m:n>	Contents of bit field of x defined by bits n through m
x<m>	M'th bit of x

Table 3-4 • Operators (Continued)

Operator	Meaning
ACCESS(x,y)	Accessibility of the location whose address is x using the access mode y. Returns a Boolean value TRUE if the address is accessible, else FALSE.
AND	Logical product
ARITH_RIGHT_SHIFT(x,y)	Arithmetic right shift of first operand by the second operand. Y is an unsigned shift value. Bit 63, the sign bit, is copied into vacated bit positions and shifted out bits are discarded.
BYTE_ZAP(x,y)	X is a quadword, y is an 8-bit vector in which each bit corresponds to a byte of the result. The y bit to x byte correspondence is y<n> ⟷ x<8n+7:8n>. This correspondence also exists between y and the result. For each bit of y from n = 0 to 7, if y <n> is 0 then byte <n> of x is copied to byte <n> of result, and if y <n> is 1 then byte <n> of result is forced to all zeros.

CASE	The CASE construct selects one of several actions based on the value of its argument. The form of a case is:
	
	`CASE argument OF`
	`  argvalue1: action_1`
	`  argvalue2: action_2`
	`  ...`
	`  argvaluen: action_n`
	`  [otherwise: default_action]`
	`ENDCASE`
	
	If the value of argument is argvalue1 then action_1 is executed; if argument = argvalue2, then action_2 is executed, and so forth.
	
	Once a single action is executed, the code stream breaks to the ENDCASE (there is an implicit break as in Pascal). Each action may nonetheless be a sequence of pseudocode operations, one operation per line.
	
	Optionally, the last argvalue may be the atom 'otherwise'. The associated default action will be taken if none of the other argvalues match the argument.

DIV	Integer division (truncates)
LEFT_SHIFT(x,y)	Logical left shift of first operand by the second operand. Y is an unsigned shift value. Zeros are moved into the vacated bit positions, and shifted out bits are discarded.

Table 3-4 • Operators (Continued)

Operator	Meaning
NOT	Logical (ones) complement
OR	Logical sum
x MOD y	x modulo y
Relational Operators	See below
MINU(x,y)	Returns the smaller of x and y, with x and y interpreted as unsigned integers
PHYSICAL_ADDRESS	Translation of a virtual address
PRIORITY_ENCODE	Returns the bit position of most significant set bit, interpreting its argument as a positive integer (= int(lg(x))) . For example: `priority_encode( 255 ) = 7`
RIGHT_SHIFT(x,y)	Logical right shift of first operand by the second operand. Y is an unsigned shift value. Zeros are moved into vacated bit positions, and shifted out bits are discarded.
SEXT(x)	X is sign-extended to the required size.
TEST(x,cond)	The contents of register x are tested for branch condition (cond) true. TEST returns a Boolean value TRUE if x bears the specified relation to 0, else FALSE is returned. Integer and floating test conditions are drawn from the preceding list of relational operators.
XOR	Logical difference
ZEXT(x)	X is zero-extended to the required size.

Operator	Meaning
LT	Less than signed
LTU	Less than unsigned
LE	Less or equal signed
LEU	Less or equal unsigned
EQ	Equal signed and unsigned
NE	Not equal signed and unsigned
GE	Greater or equal signed
GEU	Greater or equal unsigned
GT	Greater signed
GTU	Greater unsigned
LBC	Low bit clear
LBS	Low bit set

Notation Conventions

The following conventions are used:

1. Only operands that appear on the left side of a replacement operator are modified.
2. No operator precedence is assumed other than that replacement (←) has the lowest precedence. Explicit precedence is indicated by the use of "{}".
3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example, "+" applied to G_floating operands means a G_floating add, whereas "+" applied to quadword operands is an integer add. Similarly, "LT" is a G_floating comparison when applied to G_floating operands and an integer comparison when applied to quadword operands.

• Instruction Formats

There are five basic Alpha instruction formats:

- Memory
- Branch
- Operate
- Floating-point Operate
- PALcode

All instruction formats are 32 bits long with a 6-bit major opcode field in bits <31:26> of the instruction.

Any unused register field (Ra, Rb, Fa, Fb) of an instruction must be set to a value of 31.

Software Note

There are several instructions, each formatted as a memory instruction, that do not use the Ra and/or Rb fields. These instructions are: Memory Barrier, Fetch, Fetch_M, Read Process Cycle Counter, Read and Clear, Read and Set, and Trap Barrier.

Memory Instruction Format

The Memory format is used to transfer data between registers and memory, to load an effective address, and for subroutine jumps. It has the format shown in Figure 3-1.

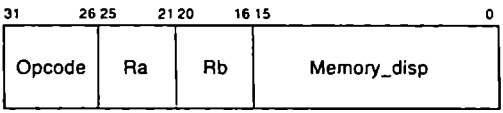


Figure 3-1 • Memory Instruction Format

A Memory format instruction contains a 6-bit opcode field, two 5-bit register address fields, Ra and Rb, and a 16-bit signed displacement field.

The displacement field is a byte offset. It is sign-extended and added to the contents of register Rb to form a virtual address. Overflow is ignored in this calculation.

The virtual address is used as a memory load/store address or a result value, depending on the specific instruction. The virtual address (va) is computed as follows for all memory format instructions except the load address high (LDAH):

```
va ← (Rbv + SEXT(Memory_disp))
```

For LDAH the virtual address (va) is computed as follows:

```
va ← (Rbv + SEXT(Memory_disp*65536))
```

Memory Format Instructions with a Function Code

Memory format instructions with a function code replace the memory displacement field in the memory instruction format with a function code that designates a set of miscellaneous instructions. The format is shown in Figure 3-2.

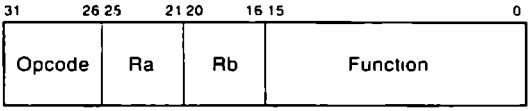


Figure 3-2 • Memory Instruction with Function Code Format

The memory instruction with function code format contains a 6-bit opcode field and a 16-bit function field. Unused function encodings produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

There are two fields, Ra and Rb. The usage of those fields depends on the instruction. See *Miscellaneous Instructions* in Chapter 4.

Memory Format Jump Instructions

For computed branch instructions (CALL, RET, JMP, JSR_COROUTINE) the displacement field is used to provide branch-prediction hints as described in *Control Instructions* in Chapter 4.

Branch Instruction Format

The Branch format is used for conditional branch instructions and for PC-relative subroutine jumps. It has the format shown in Figure 3-3.

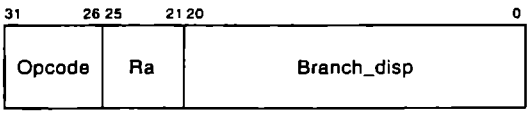


Figure 3-3 • Branch Instruction Format

A Branch format instruction contains a 6-bit opcode field, one 5-bit register address field (Ra), and a 21-bit signed displacement field.

The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as follows:

```
va ← PC + {4*SEXT(Branch_disp)}
```

Operate Instruction Format

The Operate format is used for instructions that perform integer register to integer register operations. The Operate format allows the specification of one destination operand and two source operands. One of the source operands can be a literal constant. The Operate format in Figure 3-4 shows the two cases when bit <12> of the instruction is 0 and 1.

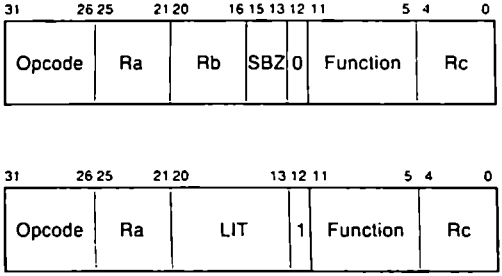


Figure 3-4 • Operate Instruction Format

An Operate format instruction contains a 6-bit opcode field and a 7-bit function field. Unused function encodings produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

There are three operand fields, Ra, Rb, and Rc.

The Ra field specifies a source operand. Symbolically, the integer Rav operand is formed as follows:

```
IF inst<25:21> EQ 31 THEN
    Rav ← 0
ELSE
    Rav ← Ra
END
```

The Rb field specifies a source operand. Integer operands can specify a literal or an integer register using bit <12> of the instruction.

If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits. Symbolically, the integer Rbv operand is formed as follows:

```
IF inst<12> EQ 1 THEN
    Rbv ← ZEXT(inst<20:13>)
ELSE
    IF inst<20:16> EQ 31 THEN
        Rbv ← 0
    ELSE
        Rbv ← Rb
    END
END
```

The Rc field specifies a destination operand.

Floating-Point Operate Instruction Format

The Floating-point Operate format is used for instructions that perform floating-point register to floating-point register operations. The Floating-point Operate format allows the specification of one destination operand and two source operands. The Floating-point Operate format is shown in Figure 3-5.

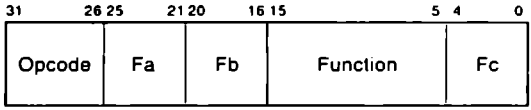


Figure 3-5 • Floating-Point Operate Instruction Format

A Floating-point Operate format instruction contains a 6-bit opcode field and an 11-bit function field. Unused function encodings produce UNPREDICTABLE results, as defined in *UNPREDICTABLE and UNDEFINED* in Chapter 1.

There are three operand fields, Fa, Fb, and Fc. Each operand field specifies either an integer or floating-point operand as defined by the instruction.

The Fa field specifies a source operand. Symbolically, the Fav operand is formed as follows:

```
IF inst<25:21> EQ 31 THEN
    Fav ← 0
ELSE
    Fav ← Fa
END
```

The Fb field specifies a source operand. Symbolically, the Fbv operand is formed as follows.

```
IF inst<20:16> EQ 31 THEN
    Fbv ← 0
ELSE
    Fbv ← Fb
END
```

Note

Neither Fa nor Fb can be a literal in Floating-point Operate instructions.

The Fc field specifies a destination operand.

Floating-Point Convert Instructions

Floating-point Convert instructions use a subset of the Floating-point Operate format and perform register-to-register conversion operations. The Fb operand specifies the source; the Fa field must be F31.

The floating-point register to be used is specified by the Fa, Fb, and Fc fields all pointing to the same floating-point register. If the Fa, Fb, and Fc fields do not all point to the same floating-point register, then it is UNPREDICTABLE which register is used.

PALcode Instruction Format

The Privileged Architecture Library (PALcode) format is used to specify extended processor functions. It has the format shown in Figure 3-6.

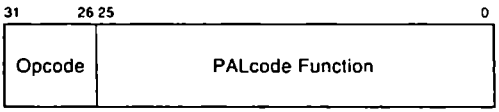


Figure 3-6 • PALcode Instruction Format

The 26-bit PALcode function field specifies the operation.

The source and destination operands for PALcode instructions are supplied in fixed registers that are specified in the individual instruction descriptions.

An opcode of zero and a PALcode function of zero specify the HALT instruction.

Chapter 4 • Instruction Descriptions

• Instruction Set Overview

This chapter describes the instructions implemented by the Alpha architecture. The instruction set is divided into the following sections:

Instruction Type	Section
Integer load and store	*Memory Integer Load/Store Instructions*
Integer control	*Control Instructions*
Integer arithmetic	*Integer Arithmetic Instructions*
Logical and shift	*Logical and Shift Instructions*
Byte manipulation	*Byte-Manipulation Instructions*
Floating-point load and store	*Memory Format Floating-Point Instructions*
Floating-point control	*Branch Format Floating-Point Instructions*
Floating-point operate	*Floating-Point Operate Format Instructions*
Miscellaneous	*Miscellaneous Instructions*

Within each major section, closely related instructions are combined into groups and described together. The instruction group description is composed of the following:

- The group name
- The format of each instruction in the group, which includes the name, access type, and data type of each instruction operand
- The operation of the instruction
- Exceptions specific to the instruction
- The instruction mnemonic and name of each instruction in the group
- Qualifiers specific to the instructions in the group
- A description of the instruction operation
- Optional programming examples and optional notes on the instruction

Subsetting Rules

An instruction that is omitted in a subset implementation of the Alpha architecture is not performed in either hardware or PALcode. System software may provide emulation routines for subsetted instructions.

Floating-Point Subsets

Floating-point support is optional on an Alpha processor. An implementation that supports floating-point must implement the 32 floating-point registers, the Floating-point Control Register (FPCR) and the instructions to access it, floating-point branch instructions, floating-point copy sign (CPYSx) instructions, floating-point convert instructions, floating-point conditional move instruction (FCMOV), and the S_floating and T_floating memory operations.

Software Note

A system that will not support floating-point operations is still required to provide the 32 floating-point registers, the Floating-point Control Register (FPCR) and the instructions to access it, and the T_floating memory operations if the system intends to support VMS. This requirement facilitates the implementation of a floating-point emulator and simplifies context-switching.

In addition, floating-point support requires at least one of the following subset groups:

1. VAX Floating-point Operate and Memory instructions (F_ and G_floating).
2. IEEE Floating-point Operate instructions (S_ and T_floating). Within this group, an implementation can choose to include or omit separately the ability to perform IEEE rounding to plus infinity and minus infinity.

Note: if one instruction in a group is provided, all other instructions in that group must be provided. An implementation with full floating-point support includes both groups; a subset floating-point implementation supports only one of these groups. The individual instruction descriptions indicate whether an instruction can be subsetted.

Software Emulation Rules

General-purpose layered and application software that executes in User mode may assume that certain loads (LDL, LDQ, LDF, LDG, LDS, and LDT) and certain stores (STL, STQ, STF, STG, STL and STT) of unaligned data are emulated by system software. General-purpose layered and application software that executes in User mode may assume that subsetted instructions are emulated by system software. Frequent use of emulation may be significantly slower than using alternative code sequences.

Emulation of loads and stores of unaligned data and subsetted instructions need not be provided in privileged access modes. System software that supports special-purpose dedicated applications need not provide emulation in User mode if emulation is not needed for correct execution of the special-purpose applications.

Opcode Qualifiers

Some Operate format and Floating-point Operate format instructions have several variants. For example, for the VAX formats, Add F_floating (ADDF) is supported with and without floating underflow enabled, and with either chopped or VAX rounding. For IEEE formats, IEEE unbiased rounding, chopped, round toward plus infinity, and round toward minus infinity can be selected.

The different variants of such instructions are denoted by opcode qualifiers, which consist of a slash (/) followed by a string of selected qualifiers. Each qualifier is denoted by a single character as shown in Table 4-1. The opcodes for each qualifier are listed in Appendix C.

Table 4-1 • Opcode Qualifiers

Qualifier	Meaning
C	Chopped rounding
D	Rounding mode dynamic
M	Round toward minus infinity
I	Inexact result enable
S	Software completion enable
U	Floating underflow enable
V	Integer overflow enable

The default values are normal rounding, software completion disabled, inexact result disabled, floating underflow disabled, and integer overflow disabled.

▪ Memory Integer Load/Store Instructions

The instructions in this section move data between the integer registers and memory.

They use the Memory instruction format. The instructions are summarized in Table 4-2.

Table 4-2 • Memory Integer Load/Store Instructions

Mnemonic	Operation
LDA	Load Address
LDAH	Load Address High
LDL	Load Sign-Extended Longword
LDL_L	Load Sign-Extended Longword Locked
LDQ	Load Quadword
LDQ_L	Load Quadword Locked
LDQ_U	Load Quadword Unaligned
STL	Store Longword
STL_C	Store Longword Conditional
STQ	Store Quadword
STQ_C	Store Quadword Conditional
STQ_U	Store Quadword Unaligned

Load Address

Format:

```
LDAx    Ra.wq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
Ra ← Rbv + SEXT(disp)                 !LDA

Ra ← Rbv + SEXT(disp*65536)           !LDAH
```

Exceptions:

None

Instruction mnemonics:

LDA	Load Address
LDAH	Load Address High

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement for LDA, and 65536 times the sign-extended 16-bit displacement for LDAH. The 64-bit result is written to register Ra.

Load Memory Data into Integer Register

Format:

```
LDx     Ra.wq,disp.ab(Rb.ab)            !Memory format
```

Operation:

```
va ←  (Rbv + SEXT(disp))

Ra  ←  SEXT((va)<31:0>)                 !LDL
Ra  ←  (va)<63:0>                       !LDQ
```

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Instruction mnemonics:

LDL	Load Sign-Extended Longword from Memory to Register
LDQ	Load Quadword from Memory to Register

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory, sign-extended, and written to register Ra. If the data is not naturally aligned, an alignment exception is generated.

Load Unaligned Memory Data into Integer Register

Format:

```
LDQ_U     Ra.wq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
va ←  ((Rbv + SEXT(disp)) AND NOT 7)

Ra  ←  (va)<63:0>
```

Exceptions:

Access Violation

Fault on Read

Translation Not Valid

Instruction mnemonics:

LDQ_U Load Unaligned Quadword from Memory to Register

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement, then the low-order three bits are cleared. The source operand is fetched from memory and written to register Ra.

Load Memory Data into Integer Register Locked

Format:

```
LDx_L       Ra.wq,disp.ab(Rb.ab)                    !Memory format
```

Operation:

```
va ← (Rbv + SEXT(disp))

lock_flag ← 1
locked_physical_address ← PHYSICAL_ADDRESS(va)
Ra ← SEXT((va)<31:0>)                               !LDL_L
Ra ← (va)<63:0>                                     !LDQ_L
```

Exceptions:

Access Violation

Alignment

Fault on Read

Translation Not Valid

Instruction mnemonics:

LDL_L	Load Sign-Extended Longword from Memory to Register Locked
LDQ_L	Load Quadword from Memory to Register Locked

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory, sign-extended for LDL_L, and written to register Ra.

When a LDx_L instruction is executed without faulting, the processor records the target physical address in a per-processor locked_physical_address register and sets the per-processor lock_flag.

If the per-processor lock_flag is (still) set when a STx_C instruction is executed, the store occurs; otherwise, it does not occur, as described for the STx_C instructions.

If processor A's lock_flag is set and processor B successfully does a store within A's locked range of physical addresses, then A's lock_flag is cleared. A processor's locked range is the aligned block of 2**N bytes that includes the locked_physical_address. The 2**N value is implementation dependent. It is at least 8 (minimum lock range is an aligned quadword) and is at most the page size for that implementation (maximum lock range is one physical page).

A processor's lock_flag is also cleared if that processor encounters any exception, interrupt, or CALL_PAL instruction. It is UNPREDICTABLE whether a processor's lock_flag is cleared by that processor's executing a normal load or store instruction. It is UNPREDICTABLE whether a processor's lock_flag is cleared by that processor's executing a taken branch (including BR, BSR, and Jumps); conditional branches that fall through do not clear the lock_flag.

The sequence LDx_L, modify, STx_C, BEQ xxx executed on a given processor does an atomic read-modify-write of a datum in shared memory if the branch falls through; if the branch is taken, the store did not modify memory and the sequence may be repeated until it succeeds.

Notes:

- LDx_L instructions do not check for write access; hence a matching STx_C may take an access-violation or fault-on-write exception.

  Executing a LDx_L instruction on one processor does not affect any architecturally visible state on another processor, and in particular cannot cause a STx_C on another processor to fail.

  LDx_L and STx_C instructions need not be paired. In particular, an LDx_L may be followed by a conditional branch: on the fall-through path an STx_C is done, whereas on the taken path no matching STx_C is done.

  If two LDx_L instructions execute with no intervening STx_C, the second one overwrites the state of the first one. If two STx_C instructions execute with no intervening LDx_L, the second one always fails because the first clears lock_flag.

- Software will not emulate unaligned LDx_L instructions.
- If any other memory access (LDx, LDQ_U, STx, STQ_U) is done on the given processor between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this.
- If a branch is taken between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this. (CMOVxx may be used to avoid branching.)
- If a subsetted instruction (for example, floating-point) is done between the LDx_L and the STx_C, the sequence above may always fail on some implementations, because of the Illegal Instruction Trap; hence, no useful program should do this.
- If a large number of instructions are executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations, because of a timer interrupt always clearing the lock_flag before the sequence completes; hence, no useful program should do this.

- Hardware implementations are encouraged to lock no more than 128 bytes. Software implementations are encouraged to separate locked locations by at least 128 bytes from other locations that could potentially be written by another processor while the first location is locked.

Implementation Notes

Implementations that impede the mobility of a cache block on LDx_L, such as that which may occur in a Read for Ownership cache coherency protocol, may release the cache block and make the subsequent STx_C fail if a branch-taken or memory instruction is executed on that processor.

All implementations should guarantee that at least 40 non-subsetted operate instructions can be executed between timer interrupts.

Store Integer Register Data into Memory Conditional

Format:

```
STx_C      Ra.wq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
va ←  (Rbv + SEXT(disp))

IF lock_flag EQ 1 THEN
   (va)<31:0> ←  Rav<31:0>              !STL_C
   (va) ←  Rav                          !STQ_C
Ra ←  lock_flag
lock_flag ←  0
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STL_C	Store Longword from Register to Memory Conditional
STQ_C	Store Quadword from Register to Memory Conditional

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. If the lock_flag is set, the Ra operand is written to memory at this address. (See the LDx_L description for conditions that clear the lock_flag.) The lock_flag is returned in RA and then set to a zero.

Notes:

- Software will not emulate unaligned STx_C instructions.
- Each implementation must do the test and store atomically, so that if two processors execute store conditionals within the same lock range, exactly one of the stores succeeds.

- The following sequence should not be used:

```
try_again: LDQ_L   R1,x
           <modify R1>
           STQ_C   R1,x
           BEQ     R1, try_again
```

That sequence penalizes performance when the STQ_C succeeds, because the sequence contains a backward branch, which is predicted to be taken in the Alpha architecture. In the case where the STQ_C succeeds and the branch will actually fall through, that sequence incurs unnecessary delay due to a mispredicted backward branch. Instead, a forward branch should be used to handle the failure case as shown in *Atomic Update of a Single Datum* in Chapter 5.

Software Note

Although this is not recommended, the address specified by a STx_C instruction need not match that given in a preceding LDx_L. Further, specifying unmatched addresses for those instructions requires an MB in between to guarantee ordering.

Implementation Notes

A STx_C must propagate to the point of coherency, where it is guaranteed to prevent any other store from changing the state of the lock bit, before its outcome can be determined.

If an implementation could encounter a TB or cache miss on the data reference of the STx_C in the sequence above (as might occur in some shared I- and D-stream direct-mapped TBs/caches), it must be able to resolve the miss and complete the store without always failing.

Store Integer Register Data into Memory

Format:

```
STx     Ra.rq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
va ←  (Rbv + SEXT(disp))
(va)<31:0> ←  Rav<31:0>               !STL
(va) ←  Rav                           !STQ
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STL	Store Longword from Register to Memory
STQ	Store Quadword from Register to Memory

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The Ra operand is written to memory at this address. If the data is not naturally aligned, an alignment exception is generated.

Store Unaligned Integer Register Data into Memory

Format:

```
STQ_U     Ra.rq,disp.ab(Rb.ab)          !Memory format
```

Operation:

```
va ← {(Rbv + SEXT(disp)) AND NOT 7}

(va)<63:0> ← Rav<63:0>
```

Exceptions:

Access Violation

Fault on Write

Translation Not Valid

Instruction mnemonics:

STQ_U	Store Unaligned Quadword from Register to Memory

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement, then clearing the low order three bits. The Ra operand is written to memory at this address.

Control Instructions

Alpha provides integer conditional branch, unconditional branch, Branch to Subroutine, and Jump to Subroutine instructions. The PC used in these instructions is the updated PC, as described in *Program Counter* in Chapter 3.

To allow implementations to achieve high performance, the Alpha architecture includes explicit hints based on a branch-prediction model:

1. For many implementations of computed branches (JSR/RET/JMP), there is a substantial performance gain in forming a good guess of the expected target I-cache address before register Rb is accessed.
2. For many implementations, the first-level (or only) I-cache is no bigger than a page (8 KB to 64 KB).
3. Correctly predicting subroutine returns is important for good performance. Some implementations will therefore keep a small stack of predicted subroutine return I-cache addresses.

The Alpha architecture provides three kinds of branch-prediction hints: likely target address, return-address stack action, and conditional branch-taken.

For computed branches (JSR/RET/JMP), otherwise unused displacement bits are used to specify the low 16 bits of the most likely target address. The PC-relative calculation using these bits can be exactly the PC-relative calculation used in unconditional branches. The low 16 bits are enough to specify an I-cache block within the largest possible Alpha page and hence are expected to be enough for branch-prediction logic to start an early I-cache access for the most likely target.

For all branches, hint or opcode bits are used to distinguish simple branches, subroutine calls, subroutine returns, and coroutine links. These distinctions allow branch-predict logic to maintain an accurate stack of predicted return addresses.

For conditional branches, the sign of the target displacement is used as a taken/fall-through hint. The instructions are summarized in Table 4-3.

Table 4-3 • Control Instructions Summary

Mnemonic	Operation
BEQ	Branch if Register Equal to Zero
BGE	Branch if Register Greater Than or Equal to Zero
BGT	Branch if Register Greater Than Zero
BLBC	Branch if Register Low Bit Is Clear
BLBS	Branch if Register Low Bit Is Set
BLE	Branch if Register Less Than or Equal to Zero
BLT	Branch if Register Less Than Zero
BNE	Branch if Register Not Equal to Zero
BR	Unconditional Branch
BSR	Branch to Subroutine
JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

Conditional Branch

Format:

```
bxx    Ra.rq,disp.al                    !Branch format
```

Operation:

```
(update PC)
va ← PC + (4*SEXT(disp))
IF TEST(Ra.v, Condition_based_on_Opcode) THEN
    PC ← va
```

Exceptions:

None

Instruction mnemonics:

BEQ	Branch if Register Equal to Zero
BGE	Branch if Register Greater Than or Equal to Zero
BGT	Branch if Register Greater Than Zero
BLBC	Branch if Register Low Bit Is Clear
BLBS	Branch if Register Low Bit Is Set
BLE	Branch if Register Less Than or Equal to Zero
BLT	Branch if Register Less Than Zero
BNE	Branch if Register Not Equal to Zero

Qualifiers:

None

Description:

Register Ra is tested. If the specified relationship is true, the PC is loaded with the target virtual address; otherwise, execution continues with the next sequential instruction.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The conditional branch instructions are PC-relative only. The 21-bit signed displacement gives a forward/backward branch distance of +/– 1M instructions.

The test is on the signed quadword integer interpretation of the register contents; all 64 bits are tested.

Notes:

- Forward conditional branches (positive displacement) are predicted to fall through. Backward conditional branches (negative displacement) are predicted to be taken. Conditional branches do not affect a predicted return address stack.

Unconditional Branch

Format:

```
BxR   Ra.wq,disp.al                    !Branch format
```

Operation:

```
{update PC}
Ra ←  PC
PC ←  PC + {4*SEXT(disp)}
```

Exceptions:

None

Instruction mnemonics:

BR	Unconditional Branch
BSR	Branch to Subroutine

Qualifiers:

None

Description:

The PC of the following instruction (the updated PC) is written to register Ra, and then the PC is loaded with the target address.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The unconditional branch instructions are PC-relative. The 21-bit signed displacement gives a forward/backward branch distance of +/– 1M instructions.

PC-relative addressability can be established by:

```
        BR  Rx,L1
L1:
```

Notes:

- BR and BSR do identical operations. They only differ in hints to possible branch-prediction logic. BSR is predicted as a subroutine call (pushes the return address on a branch-prediction stack), whereas BR is predicted as a branch (no push).

Jumps

Format:

```
mnemonic        Ra.wq,(Rb.ab),hint        !Memory format
```

Operation:

```
(update PC)
va ← Rbv AND (NOT 3)
Ra ← PC
PC ← va
```

Exceptions:

None

Instruction mnemonics:

JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

Qualifiers:

None

Description:

The PC of the instruction following the Jump instruction (the updated PC) is written to register Ra, and then the PC is loaded with the target virtual address.

The new PC is supplied from register Rb. The low two bits of Rb are ignored. Ra and Rb may specify the same register; the target calculation using the old value is done before the new value is assigned.

All Jump instructions do identical operations. They only differ in hints to possible branch-prediction logic. The displacement field of the instruction is used to pass this information. The four different "opcodes" set different bit patterns in disp<15:14>, and the hint operand sets disp<13:0>.

These bits are intended to be used as shown in Table 4-4.

Table 4-4 • Jump Instructions Branch Prediction

disp<15:14>	Meaning	Predicted Target<15:0>	Prediction Stack Action
00	JMP	PC + (4*disp<13:0>)	–
01	JSR	PC + (4*disp<13:0>)	Push PC
10	RET	Prediction stack	Pop
11	JSR_COROUTINE	Prediction stack	Pop, push PC

The design in Table 4-4 allows specification of the low 16 bits of a likely longword target address (enough bits to start a useful I-cache access early), and also allows distinguishing call from return (and from the other two less frequent operations).

Note that the above information is used only as a hint; correct setting of these bits can improve performance but is not needed for correct operation. See Appendix A for more information on branch prediction.

An unconditional long jump can be performed by:

```
JMP  R31,(Rb),hint
```

Coroutine linkage can be performed by specifying the same register in both the Ra and Rb operands. When disp<15:14> equals '10' (RET) or '11' (JSR_COROUTINE) (that is, the target address prediction, if any, would come from a predictor implementation stack), then bits <13:0> are reserved for software and must be ignored by all implementations. All encodings for bits <13:0> are used by Digital software or Reserved to Digital, as follows:

Encoding	Meaning
0000_{16}	Indicates non-procedure return
0001_{16}	Indicates procedure return
	All other encodings are reserved to Digital.

• Integer Arithmetic Instructions

The integer arithmetic instructions perform add, subtract, multiply, and signed and unsigned compare operations.

The integer instructions are summarized in Table 4-5.

Table 4-5 • Integer Arithmetic Instructions Summary

Mnemonic	Operation
ADD	Add Quadword/Longword
S4ADD	Scaled Add by 4
S8ADD	Scaled Add by 8
CMPEQ	Compare Signed Quadword Equal
CMPLT	Compare Signed Quadword Less Than
CMPLE	Compare Signed Quadword Less Than or Equal
CMPULT	Compare Unsigned Quadword Less Than
CMPULE	Compare Unsigned Quadword Less Than or Equal
MUL	Multiply Quadword/Longword
UMULH	Multiply Quadword Unsigned High
SUB	Subtract Quadword/Longword
S4SUB	Scaled Subtract by 4
S8SUB	Scaled Subtract by 8

There is no integer divide instruction. Division by a constant can be done via UMULH; division by a variable can be done via a subroutine. See Appendix A.

Longword Add

Format:

```
ADDL     Ra.rq,Rb.rq,Rc.wq          !Operate format
ADDL     Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ← SEXT( (Rav + Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDL Add Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is added to register Rb or a literal, and the sign-extended 32-bit sum is written to Rc.

The high order 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit sum. Overflow detection is based on the longword sum Rav<31:0> + Rbv<31:0> .

Scaled Longword Add

Format:

```
SxADDL    Ra.rq,Rb.rq,Rc.wq          !Operate format
SxADDL    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
CASE
  S4ADDL:  Rc ←  SEXT (((LEFT_SHIFT(Rav,2)) + Rbv)<31:0>)
  S8ADDL:  Rc ←  SEXT (((LEFT_SHIFT(Rav,3)) + Rbv)<31:0>)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4ADDL	Scaled Add Longword by 4
S8ADDL	Scaled Add Longword by 8

Qualifiers:

None

Description:

Register Ra is scaled by 4 (for S4ADDL) or 8 (for S8ADDL) and is added to register Rb or a literal, and the sign-extended 32-bit sum is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit sum.

Quadword Add

Format:

```
ADDQ    Ra.rq,Rb.rq,Rc.wq          !Operate format
ADDQ    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ←  Rav + Rbv                    ! Quadword
```

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDQ Add Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is added to register Rb or a literal, and the 64-bit sum is written to Rc.

On overflow, the least significant 64 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate carry. After adding two values, if the sum is less unsigned than either one of the inputs, there was a carry out of the most significant bit.

Scaled Quadword Add

Format:

```
SxADDQ     Ra.rq,Rb.rq,Rc.wq          !Operate format
SxADDQ     Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
CASE
  S4ADDQ:  Rc ←  LEFT_SHIFT(Rav,2) + Rbv
  S8ADDQ:  Rc ←  LEFT_SHIFT(Rav,3) + Rbv
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4ADDQ	Scaled Add Quadword by 4
S8ADDQ	Scaled Add Quadword by 8

Qualifiers:

None

Description:

Register Ra is scaled by 4 (for S4ADDQ) or 8 (for S8ADDQ) and is added to register Rb or a literal, and the 64-bit sum is written to Rc.

On overflow, the least significant 64 bits of the true result are written to the destination register.

Integer Signed Compare

Format:

```
CMPxx    Ra.rq,Rb.rq,Rc.wq        !Operate format
CMPxx    Ra.rq,#b.ib,Rc.wq        !Operate format
```

Operation:

```
IF  Rav SIGNED_RELATION Rbv  THEN
    Rc ← 1
ELSE
    Rc ← 0
```

Exceptions:

None

Instruction mnemonics:

CMPEQ	Compare Signed Quadword Equal
CMPLE	Compare Signed Quadword Less Than or Equal
CMPLT	Compare Signed Quadword Less Than

Qualifiers:

None

Description:

Register Ra is compared to Register Rb or a literal. If the specified relationship is true, the value one is written to register Rc; otherwise, zero is written to Rc.

Notes:

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.

Integer Unsigned Compare

Format:

```
CMPUxx    Ra.rq,Rb.rq,Rc.wq          !Operate format
CMPUxx    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
IF  Rav UNSIGNED_RELATION Rbv  THEN
    Rc ←  1
ELSE
    Rc ←  0
```

Exceptions:

None

Instruction mnemonics:

CMPULE	Compare Unsigned Quadword Less Than or Equal
CMPULT	Compare Unsigned Quadword Less Than

Qualifiers:

None

Description:

Register Ra is compared to Register Rb or a literal. If the specified relationship is true, the value one is written to register Rc; otherwise, zero is written to Rc.

Longword Multiply

Format:

```
MULL    Ra.rq,Rb.rq,Rc.wq           !Operate format
MULL    Ra.Rq,#b.ib,Rc.wq           !Operate format
```

Operation:

```
Rc ← SEXT ((Rav * Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

MULL Multiply Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is multiplied by register Rb or a literal, and the sign-extended 32-bit product is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit product. Overflow detection is based on the longword product Rav<31:0> * Rbv<31:0>. On overflow, the proper sign extension of the least significant 32 bits of the true result are written to the destination register.

The MULQ instruction can be used to return the full 64-bit product.

Quadword Multiply

Format:

```
MULQ    Ra.rq,Rb.rq,Rc.wq          !Operate format
MULQ    Ra.Rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc  ←  Rav * Rbv                   !MUL
```

Exceptions:

Integer Overflow

Instruction mnemonics:

MULQ Multiply Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is multiplied by register Rb or a literal, and the 64-bit product is written to register Rc. Overflow detection is based on considering the operands and the result as signed quantities. On overflow, the least significant 64 bits of the true result are written to the destination register.

The UMULH instruction can be used to generate the upper 64 bits of the 128-bit result when an overflow occurs.

Unsigned Quadword Multiply High

Format:

```
UMULH   Ra.rq,Rb.rq,Rc.wq           !Operate format
UMULH   Ra.Rq,#b.ib,Rc.wq           !Operate format
```

Operation:

```
Rc  ←  (Rav *U Rbv)<127:64>         !UMULH
```

Exceptions:

None

Instruction mnemonics:

UMULH Unsigned Multiply Quadword High

Qualifiers:

None

Description:

Register Ra and Rb or a literal are multiplied as unsigned numbers to produce a 128-bit result. The high-order 64-bits are written to register Rc.

The UMULH instruction can be used to generate the upper 64 bits of a 128-bit result as follows:

Ra and Rb are unsigned: result of UMULH

Ra and Rb are signed: (result of UMULH) - Ra<63>*Rb - Rb<63>*Ra

The MULQ instruction gives the low 64 bits of the result in either case.

Longword Subtract

Format:

```
SUBL    Ra.rq,Rb.rq,Rc.wq          !Operate format
SUBL    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ←  SEXT ((Rav - Rbv)<31:0>)
```

Exceptions:

Integer Overflow

Instruction mnemonics:

SUBL Subtract Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Rb or a literal is subtracted from register Ra, and the sign-extended 32-bit difference is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit difference. Overflow detection is based on the longword difference Rav<31:0> – Rbv<31:0>.

Scaled Longword Subtract

Format:

```
SxSUBL    Ra.rq,Rb.rq,Rc.wq          !Operate format
SxSUBL    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
CASE
  S4SUBL:  Rc ← SEXT (((LEFT_SHIFT(Rav,2)) - Rbv)<31:0>)
  S8SUBL:  Rc ← SEXT (((LEFT_SHIFT(Rav,3)) - Rbv)<31:0>)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4SUBL	Scaled Subtract Longword by 4
S8SUBL	Scaled Subtract Longword by 8

Qualifiers:

None

Description:

Register Rb or a literal is subtracted from the scaled value of register Ra, which is scaled by 4 (for S4SUBL) or 8 (for S8SUBL), and the sign-extended 32-bit difference is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit difference.

Quadword Subtract

Format:

```
SUBQ    Ra.rq,Rb.rq,Rc.wq          !Operate format
SUBQ    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ←  Rav - Rbv
```

Exceptions:

Integer Overflow

Instruction mnemonics:

SUBQ Subtract Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Rb or a literal is subtracted from register Ra, and the 64-bit difference is written to register Rc. On overflow, the least significant 64 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate borrow. If the minuend (Rav) is less unsigned than the subtrahend (Rbv), there will be a borrow.

Scaled Quadword Subtract

Format:

```
SxSUBQ    Ra.rq,Rb.rq,Rc.wq          !Operate format
SxSUBQ    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
CASE
  S4SUBQ:  Rc ←  LEFT_SHIFT(Rav,2) - Rbv
  S8SUBQ:  Rc ←  LEFT_SHIFT(Rav,3) - Rbv
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

S4SUBQ	Scaled Subtract Quadword by 4
S8SUBQ	Scaled Subtract Quadword by 8

Qualifiers:

None

Description:

Register Rb or a literal is subtracted from the scaled value of register Ra, which is scaled by 4 (for S4SUBQ) or 8 (for S8SUBQ), and the 64-bit difference is written to Rc.

• Logical and Shift Instructions

The logical instructions perform quadword Boolean operations. The conditional move integer instructions perform conditionals without a branch. The shift instructions perform left and right logical shift and right arithmetic shift. These are summarized in Table 4-6.

Table 4-6 • Logical and Shift Instructions Summary

Mnemonic	Operation
AND	Logical Product
BIC	Logical Product with Complement
BIS	Logical Sum (OR)
EQV	Logical Equivalence (XORNOT)
ORNOT	Logical Sum with Complement
XOR	Logical Difference
CMOVxx	Conditional Move Integer
SLL	Shift Left Logical
SRA	Shift Right Arithmetic
SRL	Shift Right Logical

Software Note

There is no arithmetic left shift instruction. Where an arithmetic left shift would be used, a logical shift will do. For multiplying by a small power of two in address computations, logical left shift is acceptable.

Integer multiply should be used to perform an arithmetic left shift with overflow checking.

Bit field extracts can be done with two logical shifts. Sign extension can be done with left logical shift and a right arithmetic shift.

Logical Functions

Format:

```
mnemonic    Ra.rq,Rb.rq,Rc.wq          !Operate format
mnemonic    Ra.rq,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ←  Rav AND Rbv                      !AND
Rc ←  Rav OR  Rbv                      !BIS
Rc ←  Rav XOR Rbv                      !XOR
Rc ←  Rav AND (NOT Rbv)                !BIC
Rc ←  Rav OR  (NOT Rbv)                !ORNOT
Rc ←  Rav XOR (NOT Rbv)                !EQV
```

Exceptions:

None

Instruction mnemonics:

AND	Logical Product
BIC	Logical Product with Complement
BIS	Logical Sum (OR)
EQV	Logical Equivalence (XORNOT)
ORNOT	Logical Sum with Complement
XOR	Logical Difference

Qualifiers:

None

Description:

These instructions perform the designated Boolean function between register Ra and register Rb or a literal. The result is written to register Rc.

The "NOT" function can be performed by doing an ORNOT with zero (Ra = R31).

Conditional Move Integer

Format:

```
CMOVxx   Ra.rq,Rb.rq,Rc.wq        !Operate format
CMOVxx   Ra.rq,#b.ib,Rc.wq        !Operate format
```

Operation:

```
IF TEST(Rav, Condition_based_on_Opcode) THEN

    Rc ←  Rbv
```

Exceptions:
None

Instruction mnemonics:

CMOVEQ	CMOVE if Register Equal to Zero
CMOVGE	CMOVE if Register Greater Than or Equal to Zero
CMOVGT	CMOVE if Register Greater Than Zero
CMOVLBC	CMOVE if Register Low Bit Clear
CMOVLBS	CMOVE if Register Low Bit Set
CMOVLE	CMOVE if Register Less Than or Equal to Zero
CMOVLT	CMOVE if Register Less Than Zero
CMOVNE	CMOVE if Register Not Equal to Zero

Qualifiers:
None

Description:
Register Ra is tested. If the specified relationship is true, the value Rbv is written to register Rc.

Notes:

Except that it is likely in many implementations to be substantially faster, the instruction:

```
CMOVEQ Ra,Rb,Rc
```

is exactly equivalent to:

```
        BNE   Ra,label
        OR    Rb,Rb,Rc
label:  ...
```

For example, a branchless sequence for:

```
R1=MAX(R1,R2)
```

is:

```
        CMPLT   R1,R2,R3        ! R3=1 if R1<R2
        CMOVNE  R3,R2,R1        ! Move R2 to R1 if R1<R2
```

Shift Logical

Format:

```
SxL     Ra.rq,Rb.rq,Rc.wq                !Operate format
SxL     Ra.rq,#b.ib,Rc.wq                !Operate format
```

Operation:

```
Rc ←  LEFT_SHIFT(Rav,  Rbv<5:0>)         !SLL
Rc ←  RIGHT_SHIFT(Rav, Rbv<5:0>)         !SRL
```

Exceptions:

None

Instruction mnemonics:

SLL	Shift Left Logical
SRL	Shift Right Logical

Qualifiers:

None

Description:

Register Ra is shifted logically left or right 0 to 63 bits by the count in register Rb or a literal. The result is written to register Rc. Zero bits are propagated into the vacated bit positions.

Shift Arithmetic

Format:

```
SRA     Ra.rq,Rb.rq,Rc.wq          !Operate format
SRA     Ra.rb,#b.ib,Rc.wq          !Operate format
```

Operation:

```
Rc ←  ARITH_RIGHT_SHIFT(Rav, Rbv<5:0>)
```

Exceptions:

None

Instruction mnemonics:

SRA	Shift Right Arithmetic

Qualifiers:

None

Description:

Register Ra is right shifted arithmetically 0 to 63 bits by the count in register Rb or a literal. The result is written to register Rc. The sign bit (Rav<63>) is propagated into the vacated bit positions.

• Byte-Manipulation Instructions

Alpha provides instructions for operating on byte operands within registers. These instructions allow full-width memory accesses in the load/store instructions combined with powerful in-register byte manipulation.

The instructions are summarized in Table 4-7.

Table 4-7 • Byte-Manipulation Instructions Summary

Mnemonic	Operation
CMPBGE	Compare Byte
EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low
EXTWH	Extract Word High
EXTLH	Extract Longword High
EXTQH	Extract Quadword High
INSBL	Insert Byte Low
INSWL	Insert Word Low
INSLL	Insert Longword Low
INSQL	Insert Quadword Low
INSWH	Insert Word High
INSLH	Insert Longword High
INSQH	Insert Quadword High
MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	Mask Quadword High
ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

Compare Byte

Format:

```
CMPBGE   Ra.rq,Rb.rq,Rc.wq        !Operate format
CMPBGE   Ra.rq,#b.ib,Rc.wq        !Operate format
```

Operation:

```
FOR i FROM 0 TO 7

  temp<8:0> ←  {0 ||      Rav<i*8+7:i*8>} +
               {0 || NOT Rbv<i*8+7:i*8>} + 1
  Rc<i> ←  temp<8>
END
Rc<63:8> ←  0
```

Exceptions:

None

Instruction mnemonics:

CMPBGE Compare Byte

Qualifiers:

None

Description:

CMPBGE does eight parallel unsigned byte comparisons between corresponding bytes of Rav and Rbv, storing the eight results in the low eight bits of Rc. The high 56 bits of Rc are set to zero. Bit 0 of Rc corresponds to byte 0, bit 1 of Rc corresponds to byte 1, and so forth. A result bit is set in Rc if the corresponding byte of Rav is greater than or equal to Rbv (unsigned).

Notes:

The result of CMPBGE can be used as an input to ZAP and ZAPNOT.

To scan for a byte of zeros in a character string:

```
        <initialize R1 to aligned QW address of string>
LOOP:
        LDQ     R2,0(R1)        ; Pick up 8 bytes
        LDA     R1,8(R1)        ; Increment string pointer
        CMPBGE  R31,R2,R3       ; If NO bytes of zero, R3<7:0>=0
        BEQ     R3,LOOP         ; Loop if no terminator byte found
        ...                     ; At this point, R3 can be used to
                                ; determine which byte terminated
```

To compare two character strings for greater/less:

```
        <initialize R1 to aligned QW address of string1>
        <initialize R2 to aligned QW address of string2>
LOOP:
        LDQ     R3,0(R1)        ; Pick up 8 bytes of string1
        LDA     R1,8(R1)        ; Increment string1 pointer
        LDQ     R4,0(R2)        ; Pick up 8 bytes of string2
        LDA     R2,8(R2)        ; Increment string2 pointer
        XOR     R3,R4,R5        ; Test for all equal bytes
        BEQ     R5,LOOP         ; Loop if all equal
        CMPBGE  R31,R5,R5       ;
        ...                     ; At this point, R5 can be used to
                                ; determine the first not-equal
                                ; byte position.
```

To range-check a string of characters in R1 for '0'..'9':

```
        LDQ     R2,lit0s        ; Pick up 8 bytes of the character
                                ; BELOW '0'  '////////'
        LDQ     R3,lit9s        ; Pick up 8 bytes of the character
                                ; ABOVE '9'  ':::::::::'
        CMPBGE  R2,R1,R4        ; Some R4<i>=1 if character is LT '0'
        CMPBGE  R1,R3,R5        ; Some R5<i>=1 if character is GT '9'
        BNE     R4,ERROR        ; Branch if some char too low
        BNE     R5,ERROR        ; Branch if some char too high
```

Extract Byte

Format:

```
EXTxx   Ra.rq,Rb.rq,Rc.wq       !Operate format
EXTxx   Ra.rq,#b.ib,Rc.wq       !Operate format
```

Operation:

```
CASE

EXTBL: byte_mask ← 0000 0001₂
EXTWx: byte_mask ← 0000 0011₂
EXTLx: byte_mask ← 0000 1111₂
EXTQx: byte_mask ← 1111 1111₂
ENDCASE

CASE

EXTxL:
    byte_loc ← Rbv<2:0>*8
    temp ← RIGHT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask) )

EXTxH:
    byte_loc ← 64 - Rbv<2:0>*8
    temp ← LEFT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask) )
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low
EXTWH	Extract Word High
EXTLH	Extract Longword High
EXTQH	Extract Quadword High

Qualifiers:

None

Description:

EXTxL shifts register Ra right by 0 to 7 bytes, inserts zeros into vacated bit positions, and then extracts 1, 2, 4, or 8 bytes into register Rc. EXTxH shifts register Ra left by 0 to 7 bytes, inserts zeros into vacated bit positions, and then extracts 2, 4, or 8 bytes into register Rc. The number of bytes to shift is specified by Rbv<2:0>. The number of bytes to extract is specified in the function code. Remaining bytes are filled with zeros.

Notes:

The comments in the examples below assume that the effective address (ea) of X(R11) is such that (ea mod 8) = 5 , the value of the aligned quadword containing X(R11) is CBAx xxxx , and the value of the aligned quadword containing X+7(R11) is yyyH GFED .

The examples below are the most general case unless otherwise noted; if more information is known about the value or intended alignment of X, shorter sequences can be used.

The intended sequence for loading a quadword from unaligned address X(R11) is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U   R2,X+7(R11)     ; Ignores va<2:0>, R2 = yyyH GFED
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTQL   R1,R3,R1        ; R1 = 0000 0CBA
EXTQH   R2,R3,R2        ; R2 = HGFE D000
OR      R2,R1,R1        ; R1 = HGFE DCBA
```

The intended sequence for loading and zero-extending a longword from unaligned address X is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U   R2,X+3(R11)     ; Ignores va<2:0>, R2 = yyyy yyyD
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTLL   R1,R3,R1        ; R1 = 0000 0CBA
EXTLH   R2,R3,R2        ; R2 = 0000 D000
OR      R2,R1,R1        ; R1 = 0000 DCBA
```

The intended sequence for loading and sign-extending a longword from unaligned address X is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U   R2,X+3(R11)     ; Ignores va<2:0>, R2 = yyyy yyyD
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTLL   R1,R3,R1        ; R1 = 0000 0CBA
EXTLH   R2,R3,R2        ; R2 = 0000 D000
OR      R2,R1,R1        ; R1 = 0000 DCBA
SLL     R1,#32,R1       ; R1 = DCBA 0000
SRA     R1,#32,R1       ; R1 = ssss DCBA
```

The intended sequence for loading and zero-extending a word from unaligned address X is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = yBAx xxxx
LDQ_U   R2,X+1(R11)     ; Ignores va<2:0>, R2 = yBAx xxxx
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTWL   R1,R3,R1        ; R1 = 0000 00BA
EXTWH   R2,R3,R2        ; R2 = 0000 0000
OR      R2,R1,R1        ; R1 = 0000 00BA
```

The intended sequence for loading and sign-extending a word from unaligned address X is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = yBAx xxxx
LDQ_U   R2,X+1(R11)     ; Ignores va<2:0>, R2 = yBAx xxxx
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTWL   R1,R3,R1        ; R1 = 0000 00BA
EXTWH   R2,R3,R2        ; R2 = 0000 0000
OR      R2,R1,R1        ; R1 = 0000 00BA
SLL     R1,#48,R1       ; R1 = BA00 0000
SRA     R1,#48,R1       ; R1 = ssss ssBA
```

The intended sequence for loading and zero-extending a byte from address X is:

```
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = yyAx xxxx
LDA     R3,X(R11)       ; R3<2:0> = (X mod 8) = 5
EXTBL   R1,R3,R1        ; R1 = 0000 000A
```

The intended sequence for loading and sign-extending a byte from address X is:

```
LDQ_U   R1, X(R11)      ; Ignores va<2:0>, R1 = yyAx xxxx
LDA     R3, X+1(R11)    ; R3<2:0> = (X + 1) mod 8, i.e.,
                        ; convert byte position within
                        ; quadword to one-origin based
EXTQH   R1, R3, R1      ; Places the desired byte into byte 7
                        ; of R1.final by left shifting
                        ; R1.initial by ( 8 - R3<2:0> ) byte
                        ; positions
SRA     R1, #56, R1     ; Arithmetic Shift of byte 7 down
                        ; into byte 0,
```

Optimized examples:

Assume that a word fetch is needed from 10(R3), where R3 is intended to contain a longword-aligned address. The optimized sequences below take advantage of the known constant offset, and the longword alignment (hence a single aligned longword contains the entire word). The sequences generate a Data Alignment Fault if R3 does not contain a longword-aligned address.

The intended sequence for loading and zero-extending an aligned word from 10(R3) is:

```
LDL     R1,8(R3)        ; R1 = ssss BAxx
                        ; Faults if R3 is not longword aligned
EXTWL   R1,#2,R1        ; R1 = 0000 00BA
```

The intended sequence for loading and sign-extending an aligned word from 10(R3) is:

```
LDL     R1,8(R3)        ; R1 = ssss BAxx
                        ; Faults if R3 is not longword aligned
SRA     R1,#16,R1       ; R1 = ssss ssBA
```

Byte Insert

Format:

```
INSxx    Ra.rq,Rb.rq,Rc.wq        !Operate format
INSxx    Ra.rq,#b.ib,Rc.wq        !Operate format
```

Operation:

```
CASE
    INSBL: byte_mask ←  0000 0000 0000 0001₂
    INSWx: byte_mask ←  0000 0000 0000 0011₂
    INSLx: byte_mask ←  0000 0000 0000 1111₂
    INSQx: byte_mask ←  0000 0000 1111 1111₂
ENDCASE
byte_mask ←  LEFT_SHIFT(byte_mask, rbv<2:0>)

CASE

    INSxL:
        byte_loc ←  Rbv<2:0>*8
        temp ←  LEFT_SHIFT(Rav, byte_loc<5:0>)
        Rc ←  BYTE_ZAP(temp, NOT(byte_mask<7:0>))
    INSxH:
        byte_loc ←  64 - Rbv<2:0>*8
        temp ←  RIGHT_SHIFT(Rav, byte_loc<5:0>)
        Rc ←  BYTE_ZAP(temp, NOT(byte_mask<15:8>))

ENDCASE
```

Exceptions:

None

Instruction mnemonics:

INSBL	Insert Byte Low
INSWL	Insert Word Low
INSLL	Insert Longword Low
INSQL	Insert Quadword Low
INSWH	Insert Word High
INSLH	Insert Longword High
INSQH	Insert Quadword High

Qualifiers:

None

Description:

INSxL and INSxH shift bytes from register Ra and insert them into a field of zeros, storing the result in register Rc. Register Rb<2:0> selects the shift amount, and the function code selects the maximum field width: 1, 2, 4, or 8 bytes. The instructions can generate a byte, word, longword, or quadword datum that is spread across two registers at an arbitrary byte alignment.

Byte Mask

Format:

```
MSKxx    Ra.rq,Rb.rq,Rc.wq        !Operate format
MSKxx    Ra.rq,#b.ib,Rc.wq        !Operate format
```

Operation:

```
CASE

    MSKBL: byte_mask ←  0000 0000 0000 0001₂
    MSKWx: byte_mask ←  0000 0000 0000 0011₂
    MSKLx: byte_mask ←  0000 0000 0000 1111₂
    MSKQx: byte_mask ←  0000 0000 1111 1111₂
ENDCASE
byte_mask ←  LEFT_SHIFT(byte_mask, rbv<2:0>)

CASE

    MSKxL:
        Rc ←  BYTE_ZAP(Rav, byte_mask<7:0>)

    MSKxH:
        Rc ←  BYTE_ZAP(Rav, byte_mask<15:8>)

ENDCASE
```

Exceptions:

None

Instruction mnemonics:

MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	Mask Quadword High

Qualifiers:

None

Description:

MSKxL and MSKxH set selected bytes of register Ra to zero, storing the result in register Rc. Register Rb<2:0> selects the starting position of the field of zero bytes, and the function code selects the maximum width: 1, 2, 4, or 8 bytes. The instructions generate a byte, word, longword, or quadword field of zeros that can spread across two registers at an arbitrary byte alignment.

Notes:

The comments in the examples below assume that the effective address (ea) of X(R11) is such that (ea mod 8) = 5, the value of the aligned quadword containing X(R11) is CBAx xxxx , the value of the aligned quadword containing X+7(R11) is yyyH GFED , and the value to be stored from R5 is hgfe dcba .

The examples below are the most general case; if more information is known about the value or intended alignment of X, shorter sequences can be used.

The intended sequence for storing an unaligned quadword R5 at address X(R11) is:

```
LDA     R6,X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U   R2,X+7(R11)     ; Ignores va<2:0>, R2 = yyyH GFED
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = CBAx xxxx
INSQH   R5,R6,R4        ; R4 = 000h gfed
INSQL   R5,R6,R3        ; R3 = cba0 0000
MSKQH   R2,R6,R2        ; R2 = yyy0 0000
MSKQL   R1,R6,R1        ; R1 = 000x xxxx
OR      R2,R4,R2        ; R2 = yyyh gfed
OR      R1,R3,R1        ; R1 = cbax xxxx
STQ_U   R2,X+7(R11)     ; Must store high then low for
STQ_U   R1,X(R11)       ; degenerate case of aligned QW
```

The intended sequence for storing an unaligned longword R5 at X is:

```
LDA     R6,X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U   R2,X+3(R11)     ; Ignores va<2:0>, R2 = yyyy yyyD
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = CBAx xxxx
INSLH   R5,R6,R4        ; R4 = 0000 000d
INSLL   R5,R6,R3        ; R3 = cba0 0000
MSKLH   R2,R6,R2        ; R2 = yyyy yyy0
MSKLL   R1,R6,R1        ; R1 = 000x xxxx
OR      R2,R4,R2        ; R2 = yyyy yyyd
OR      R1,R3,R1        ; R1 = cbax xxxx
STQ_U   R2,X+3(R11)     ; Must store high then low for
STQ_U   R1,X(R11)       ; degenerate case of aligned
```

The intended sequence for storing an unaligned word R5 at X is:

```
LDA     R6,X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U   R2,X+1(R11)     ; Ignores va<2:0>, R2 = yBAx xxxx
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = yBAx xxxx
INSWH   R5,R6,R4        ; R4 = 0000 0000
INSWL   R5,R6,R3        ; R3 = 0ba0 0000
MSKWH   R2,R6,R2        ; R2 = yBAx xxxx
MSKWL   R1,R6,R1        ; R1 = y00x xxxx
OR      R2,R4,R2        ; R2 = yBAx xxxx
OR      R1,R3,R1        ; R1 = ybax xxxx
STQ_U   R2,X+1(R11)     ; Must store high then low for
STQ_U   R1,X(R11)       ; degenerate case of aligned
```

The intended sequence for storing a byte R5 at X is:

```
LDA     R6,X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U   R1,X(R11)       ; Ignores va<2:0>, R1 = yyAx xxxx
INSBL   R5,R6,R3        ; R3 = 00a0 0000
MSKBL   R1,R6,R1        ; R1 = yy0x xxxx
OR      R1,R3,R1        ; R1 = yyax xxxx
STQ_U   R1,X(R11)       ;
```

Zero Bytes

Format:

```
ZAPx    Ra.rq,Rb.rq,Rc.wq       !Operate format
ZAPx    Ra.rq,#b.ib,Rc.wq       !Operate format
```

Operation:

```
CASE

    ZAP:
        Rc ←  BYTE_ZAP(Rav, rbv<7:0>)

    ZAPNOT:
        Rc ←  BYTE_ZAP(Rav, NOT rbv<7:0>)
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

Qualifiers:

None

Description:

ZAP and ZAPNOT set selected bytes of register Ra to zero, and store the result in register Rc. Register Rb<7:0> selects the bytes to be zeroed; bit 0 of Rbv corresponds to byte 0, bit 1 of Rbv corresponds to byte 1, and so on. A result byte is set to zero if the corresponding bit of Rbv is a one for ZAP and a zero for ZAPNOT.

• Floating-Point Instructions

Alpha provides instructions for operating on floating-point operands in each of four data formats:

- F_floating (VAX single)
- G_floating (VAX double, 11-bit exponent)
- S_floating (IEEE single)
- T_floating (IEEE double, 11-bit exponent)

Data conversion instructions are also provided to convert operands between floating-point and quadword integer formats, between double and single floating, and between quadword and longword integers.

Note

D_floating is a partially supported datatype; no D_floating arithmetic operations are provided in the architecture. For backward compatibility, exact D_floating arithmetic may be provided via software emulation. D_floating "format compatibility," in which binary files of D_floating numbers may be processed but without the last 3 bits of fraction precision, can be obtained via conversions to G_floating, G arithmetic operations, then conversion back to D_floating.

The choice of data formats is encoded in each instruction. Each instruction also encodes the choice of rounding mode and the choice of trapping mode.

All floating-point operate instructions (that is, *not* including loads or stores) that yield an F_ or G_floating zero result must materialize a true zero.

Floating Subsets and Floating Faults

All floating-point operations may take floating disabled faults. Any subsetted floating-point instruction may take an Illegal Instruction Trap. These faults are not explicitly listed in the description of each instruction.

All floating-point loads and stores may take memory management faults (access control violation, translation not valid, fault on read/write, data alignment).

The Floating-point Enable (FEN) internal processor register (IPR) allows system software to restrict access to the floating registers.

If a floating instruction is implemented and FEN = 0 , attempts to execute the instruction cause a floating disabled fault.

If a floating instruction is not implemented, attempts to execute the instruction cause an Illegal Instruction Trap. This rule holds regardless of the value of FEN.

An Alpha implementation may provide both VAX and IEEE floating-point operations, either, or none.

Some floating-point instructions are common to the VAX and IEEE subsets, some are VAX only, and some are IEEE only. These are designated in the descriptions that follow. If either subset is implemented, all the common instructions must be implemented.

An implementation including IEEE floating-point may subset the ability to perform rounding to plus infinity and minus infinity. If not implemented, instructions requesting these rounding modes take Illegal Instruction Trap.

Definitions

The following definitions apply to Alpha floating-point support.

true result
The mathematically correct result of an operation, assuming that the input operand values are exact. The true result is typically rounded to the nearest representable result.

representable result
a real number that can be represented exactly as a VAX or IEEE floating-point number, with finite precision and bounded exponent range.

LSB
The least significant bit. For a positive representable number A whose fraction is not all ones, A + 1 LSB is the next larger representable number, and A + 1/2 LSB is exactly halfway between A and the next larger representable number.

true zero
The value +0, represented as exactly 64 zeros in a floating-point register.

Alpha finite number
A floating-point number with a definite, in-range value. Specifically, all numbers in the inclusive ranges −MAX..−MIN, zero, +MIN..+MAX, where MAX is the largest non-infinite representable floating-point number and MIN is the smallest non-zero representable normalized floating-point number.

For VAX floating-point, finites do not include reserved operands or dirty zeros (this differs from the usual VAX interpretation of dirty zeros as finite). For IEEE floating-point, finites do not include infinites, NaNs, or denormals, but do include minus zero.

Not-a-Number
An IEEE floating-point bit pattern that represents something other than a number. This comes in two forms: signaling NaNs (for Alpha, those with an initial fraction bit of 1) and quiet NaNs (for Alpha, those with initial fraction bit of 0).

infinity
An IEEE floating-point bit pattern that represents plus or minus infinity.

denormal
An IEEE floating-point bit pattern that represents a number whose magnitude lies between zero and the smallest finite number.

dirty zero
A VAX floating-point bit pattern that represents a zero value, but not in true-zero form.

reserved operand
A VAX floating-point bit pattern that represents an illegal value.

trap shadow
The set of instructions potentially executed after an instruction that signals an arithmetic trap but before the trap is actually taken.

Encodings

Floating-point numbers are represented with three fields: sign, exponent, and fraction. The sign is 1 bit; the exponent is 8 or 11 bits; and the fraction is 23, 52, or 55 bits. Some encodings represent special values:

Sign	Exponent	Fraction	VAX Meaning	VAX Finite	IEEE Meaning	IEEE Finite
x	All-1's	Non-zero	Finite	Yes	+/–NaN	No
x	All-1's	0	Finite	Yes	+/–Infinity	No
0	0	Non-zero	Dirty zero	No	+Denormal	No
1	0	Non-zero	Resv. operand	No	–Denormal	No
0	0	0	True zero	Yes	+0	Yes
1	0	0	Resv. operand	No	–0	Yes
x	Other	x	Finite	Yes	finite	Yes

The values of MIN and MAX for each of the four floating-point data formats are:

Data Format	MIN	MAX
F_floating	2**–127 * 0.5 (0.294e–38)	2**127 * (1.0 – 2**–24) (1.70e38)
G_floating	2**–1023 * 0.5 (0.56e–308)	2**1023 * (1.0 – 2**–53) (0.899e308)
S_floating	2**–126 * 1.0 (1.175e–38)	2**127 * (2.0 – 2**–23) (3.40e38)
T_floating	2**–1022 * 1.0 (2.225e–308)	2**1023 * (2.0 – 2**–52) (1.798e308)

Floating-Point Rounding Modes

All rounding modes map a true result that is exactly representable to that representable value.

VAX Rounding Modes

For VAX floating-point operations, two rounding modes are provided and are specified in each instruction: normal (biased) rounding and chopped rounding.

Normal VAX rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the larger in absolute value (sometimes called biased rounding away from zero); maps true results ≥ MAX + 1/2 LSB in magnitude to an overflow; and maps true results < MIN − 1/2 LSB in magnitude to an underflow.

Chopped VAX rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results ≥ MAX + 1 LSB in magnitude to an overflow; and maps true results < MIN in magnitude to an underflow.

IEEE Rounding Modes

For IEEE floating-point operations, four rounding modes are provided: normal rounding (unbiased round to nearest), rounding toward minus infinity, round toward zero, and rounding toward plus infinity. The first three can be specified in the instruction. Rounding toward plus infinity can be obtained by setting the Floating-point Control Register (FPCR) to select it and then specifying dynamic rounding mode in the instruction (See *FPCR Register and Dynamic Rounding Mode* in this chapter). Alpha IEEE arithmetic does rounding before detecting overflow/underflow.

Normal IEEE rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the one whose fraction ends in 0 (sometimes called unbiased rounding to even); maps true results ≥ MAX + 1/2 LSB in magnitude to an overflow; and maps true results < MIN − 1/2 LSB in magnitude to an underflow.

Plus infinity IEEE rounding maps the true result to the larger of two surrounding representable results; maps true results > MAX in magnitude to an overflow; maps positive true results ≤ +MIN − 1 LSB to an underflow; and maps negative true results > −MIN to an underflow.

Minus infinity IEEE rounding maps the true result to the smaller of two surrounding representable results; maps true results > MAX in magnitude to an overflow; maps positive true results < +MIN to an underflow; and maps negative true results ≥ −MIN + 1 LSB to an underflow.

Chopped IEEE rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results ≥ MAX + 1 LSB in magnitude to an overflow; and maps non-zero true results < MIN in magnitude to an underflow.

Dynamic rounding mode uses the IEEE rounding mode selected by the FPCR register and is described in more detail in *FPCR Register and Dynamic Rounding Mode* in this chapter.

The following tables summarize the floating-point rounding modes:

VAX Rounding Mode	Instruction Notation
Normal rounding	(No modifier)
Chopped	/C

IEEE Rounding Mode	Instruction Notation
Normal rounding	(No modifier)
Dynamic rounding	/D
Plus infinity	/D and ensure that FPCR<DYN> = '11'
Minus infinity	/M
Chopped	/C

Floating-Point Trapping Modes

There are six exceptions that can be generated by floating-point operate instructions, all signaled by an arithmetic exception trap. These exceptions are:

- Invalid operation
- Division by zero
- Overflow
- Underflow, may be disabled
- Inexact result, may be disabled
- Integer overflow (conversion to integer only), may be disabled

VAX Trapping Modes

For VAX floating-point operations other than CVTxQ, four trapping modes are provided. They specify software completion and whether traps are enabled for underflow.

For VAX conversions from floating-point to integer, four trapping modes are provided. They specify software completion and whether traps are enabled for integer overflow.

IEEE Trapping Modes

For IEEE floating-point operations other than CVTxQ, four trapping modes are provided. They specify software completion and whether traps are enabled for underflow and inexact results.

For IEEE conversions from floating-point to integer, four trapping modes are provided. They specify software completion, and whether traps are enabled for integer overflow and inexact results.

The modes and instruction notation are:

VAX Trap Mode	Instruction Notation
Imprecise, underflow disabled	(No modifier)
Imprecise, underflow enabled	/U
Software, underflow disabled	/S
Software, underflow enabled	/SU

VAX Convert-to-Integer Trap Mode	Instruction Notation
Imprecise, integer overflow disabled	(No modifier)
Imprecise, integer overflow enabled	/V
Software, integer overflow disabled	/S
Software, integer overflow enabled	/SV

IEEE Trap Mode	Instruction Notation
Imprecise, unfl disabled, inexact disabled	(No modifier)
Imprecise, unfl enabled, inexact disabled	/U
Software, unfl enabled, inexact disabled	/SU
Software, unfl enabled, inexact enabled	/SUI

IEEE Convert-to-Integer Trap Mode	Instruction Notation
Imprecise, int.ovfl disabled, inexact disabled	(No modifier)
Imprecise, int.ovfl enabled, inexact disabled	/V
Software, int.ovfl enabled, inexact disabled	/SV
Software, int.ovfl enabled, inexact enabled	/SVI

Imprecise /Software Completion Trap Modes

Floating-point instructions may be pipelined, and all exceptions are imprecise traps:

- The trapping instruction may write an UNPREDICTABLE result value.
- The trap PC is an arbitrary number of instructions past the one triggering the trap. The trigger instruction plus all intervening executed instructions are collectively referred to as the *trap shadow* of the trigger instruction.
- The extent of the trap shadow is bounded only by a TRAPB instruction (or the implicit TRAPB within a CALL_PAL instruction).
- Input operand values may have been overwritten in the trap shadow.
- Result values may have been overwritten in the trap shadow.
- An UNPREDICTABLE result value may have been used as an input operand in the trap shadow.
- Additional traps may occur in the trap shadow.
- In general, it is not feasible to fix up the result value or to continue from the trap.

This behavior is ideal for operations on finite operands that give finite results. For programs that deliberately operate outside the overflow/underflow range, or use IEEE NaNs, software assistance is required to complete floating-point operations correctly. This assistance can be provided by a software arithmetic trap handler, plus constraints on the instructions surrounding the trap.

For a trap handler to complete non-finite arithmetic, the following conditions must hold:

1. On entry to the trap shadow, if any Alpha register or memory location contains a value that is used as an operand value by some instruction in the trap shadow (live on entry), then no instruction in the trap shadow may modify the register or memory location.
2. Within the trap shadow, the computation of the base register for a memory load or store instruction may not involve using the result of an instruction that might generate an UNPREDICTABLE result.
3. Within the trap shadow, no register may be used more than once as a destination register.
4. The trap shadow may not include any branch instructions.
5. Each floating instruction to be completed must be so marked, by specifying the /S software completion modifier.

The first condition allows a software trap handler to emulate the trigger instruction with its original input operand values and then to reexecute the rest of the trap shadow.

The second condition prevents memory accesses at unpredictable addresses.

The remaining conditions make it possible for a software trap handler to find the trigger instruction via a linear scan backwards from the trap PC.

Note

The /S modifier does not affect instruction operation or trap behavior; it is an informational bit passed to a software trap handler. It allows a trap handler to test easily whether an instruction is intended to be completed. (The /S bits of instructions signaling traps are carried into the trap summary.) The handler may then assume that the other conditions are met without examining the code stream.

If a software trap handler is provided, it must handle the completion of all floating-point operations marked /S that follow the rules above. In effect, one TRAPB instruction per basic block can be used.

Invalid Operation Arithmetic Trap

An invalid operation arithmetic trap is signaled if any operand of a floating arithmetic-operate instruction is non-finite. (CMPTxy is an exception to the rule and operates normally with plus and minus infinity and does not trap in this case.) This trap is always enabled. If this trap occurs, an UNPREDICTABLE value is stored in the result register. (IEEE-compliant system software must also supply an invalid operation indication to the user for SQRT of a negative non-zero number, 0/0, × REM 0 , and conversions to integer that take an integer overflow trap.)

Division by Zero Arithmetic Trap

A division by zero arithmetic trap is taken if the numerator does not cause an invalid operation trap and the denominator is zero. This trap is always enabled. If this trap occurs, an UNPREDICTABLE value is stored in the result register.

Overflow Arithmetic Trap

An overflow arithmetic trap is signaled if the rounded result exceeds in magnitude the largest finite number of the destination format. This trap is always enabled. If this trap occurs, an UNPREDICTABLE value is stored in the result register.

Underflow Arithmetic Trap

An underflow occurs if the rounded result is smaller in magnitude than the smallest finite number of the destination format.

If an underflow occurs, a true zero (64 bits of zero) is always stored in the result register, even if the proper IEEE result would have been –0 (underflow below the negative denormal range).

If an underflow occurs and underflow traps are enabled by the instruction, an underflow arithmetic trap is signaled.

Inexact Result Arithmetic Trap

An inexact result occurs if the infinitely precise result differs from the rounded result.

If an inexact result occurs, the normal rounded result is still stored in the result register.

If an inexact result occurs and inexact result traps are enabled by the instruction, an inexact result arithmetic trap is signaled.

Integer Overflow Arithmetic Trap

In conversions from floating to quadword integer, an integer overflow occurs if the rounded result is outside the range –2**63..2**63–1 . In conversions from quadword integer to longword integer, an integer overflow occurs if the result is outside the range –2**31..2**31–1 .

If an integer overflow occurs in CVTxQ or CVTQL, the true result truncated to the low-order 64 or 32 bits respectively is stored in the result register.

If an integer overflow occurs and integer overflow traps are enabled by the instruction, an integer overflow arithmetic trap is signaled.

Floating-Point Single-Precision Operations

Single-precision values (F_floating or S_floating) are stored in the floating registers in canonical form, as subsets of double-precision values, with 11-bit exponents restricted to the corresponding single-precision range, and with the 29 low-order fraction bits restricted to be all zero.

Single-precision operations applied to canonical single-precision values give single-precision results. Single-precision operations applied to non-canonical operands give UNPREDICTABLE results.

Longword integer values in floating registers are stored in bits <63:62,58:29>, with bits <61:59> ignored and zeros in bits <28:0>.

FPCR Register and Dynamic Rounding Mode

When an IEEE floating-point operate instruction specifies dynamic mode (/D) in its function field (function code bits <7:6> = 11), the rounding mode to be used for the instruction is derived from the FPCR register. The layout of the rounding mode bits and their assignments matches exactly the format used in the 11-bit function field of the floating-point operate instructions.

In addition, the FPCR gives a summary for each exception type of the exceptions conditions detected by all IEEE floating-point operates thus far as well as an overall summary bit that indicates whether any of these exception conditions has been detected. The individual exception bits match exactly in purpose and order the exceptions bits found in the exception summary quadword that is pushed for arithmetic traps. However, for each instruction, these exceptions bits are set independent of the trapping mode specified for the instruction. Therefore, even though trapping may be disabled for a certain exceptional condition, the fact that the exceptional condition was encountered by an instruction will still be recorded in the FPCR.

Floating-point operates that belong to the IEEE subset and CVTQL, which belongs to both VAX and IEEE subsets, appropriately set the FPCR exception bits. It is UNPREDICTABLE whether floating-point operates that belong only to the VAX floating-point subset set the FPCR exception bits.

Alpha floating-point hardware only transitions these exception bits from zero to one. Once set to one, these exception bits are only cleared when software writes zero into these bits by writing a new value into the FPCR.

The format of the FPCR is shown in Figure 4-1 and described in Table 4-8.

Figure 4-1 • Floating-Point Control Register (FPCR) Format

Table 4-8 • Floating-Point Control Register (FPCR) Bit Descriptions

Bit	Description
63	Summary Bit (SUM). Records bitwise OR of FPCR exception bits. Equal to (FPCR[57] | FPCR[56] | FPCR[55] | FPCR[54] | FPCR[53] | FPCR[52]).
62–60	Reserved. Read As Zero; Ignored when written.
59–58	Dynamic Rounding Mode (DYN). Indicates the rounding mode to be used by an IEEE floating-point operate instruction when the instruction's function field specifies dynamic mode (/D). Assignments are:

DYN	IEEE Rounding Mode Selected
00	Chopped rounding mode
01	Minus infinity
10	Normal rounding
11	Plus infinity

57	Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.
56	Inexact Result (INE). A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
55	Underflow (UNF). A floating arithmetic or conversion operation underflowed the destination exponent.
54	Overflow (OVF). A floating arithmetic or conversion operation overflowed the destination exponent.
53	Division by Zero (DZE). An attempt was made to perform a floating divide operation with a divisor of zero.
52	Invalid Operation (INV). An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.
51–0	Reserved. Read As Zero; Ignored when written.

FPCR is read from and written to the floating-point registers by the MT_FPCR and MF_FPCR instructions respectively, which are described in *Accessing the FPCR* in this chapter.

FPCR and the instructions to access it are required for an implementation that supports floating-point (see *Floating-Point Subsets* in this chapter). On implementations that do not support floating-point, the instructions that access FPCR (MF_FPCR and MT_FPCR) take an Illegal Instruction Trap.

Software Note

As noted in *Floating-Point Subsets* in this chapter, support for FPCR is required on a system that supports VMS even if that system does not support floating-point.

Accessing the FPCR

Because Alpha floating-point hardware can overlap the execution of a number of floating-point instructions, accessing the FPCR must be synchronized with other floating-point instructions. A TRAPB must be issued both prior to and after accessing the FPCR to ensure that the FPCR access is synchronized with the execution of previous and subsequent floating-point instructions; otherwise synchronization is not ensured.

Issuing a TRAPB followed by an MT_FPCR followed by another TRAPB ensures that only floating-point instructions issued after the second TRAPB are affected by and affect the new value of the FPCR. Issuing a TRAPB followed by an MF_FPCR followed by another TRAPB ensures that the value read from the FPCR only records the exception information for floating-point instructions issued prior to the first TRAPB.

Consider the following example:

```
ADDT/D
TRAPB                   ;1
MT_FPCR F1,F1,F1
TRAPB                   ;2
SUBT/D
```

Without the first TRAPB, it is possible in an implementation for the ADDT/D to execute in parallel with the MT_FPCR. Thus, it would be UNPREDICTABLE whether the ADDT/D was affected by the new rounding mode set by the MT_FPCR and whether fields cleared by the MT_FPCR in the exception summary were subsequently set by the ADDT/D.

Without the second TRAPB, it is possible in an implementation for the MT_FPCR to execute in parallel with the SUBT/D. Thus, it would be UNPREDICTABLE whether the SUBT/D was affected by the new rounding mode set by the MT_FPCR and whether fields cleared by the MT_FPCR in the exception summary field of FPCR were previously set by the SUBT/D.

Default Values of the FPCR

Processor initialization leaves the value of FPCR UNPREDICTABLE.

Software Note

Digital software should initialize FPCR<DYN> = 11 during program activation. Using this default, interval arithmetic code can switch from plus to minus infinity rounding with no penalty in performance by using /M and /D qualifiers.

Program activation should clear all other fields of the FPCR.

Saving and Restoring the FPCR

The FPCR must be saved and restored across context switches so that the FPCR value of one process does not affect the rounding behavior and exception summary of another process.

The dynamic rounding mode put into effect by the programmer (or initialized by image activation) is valid for the entirety of the program and remains in effect until subsequently changed by the programmer or until image run-down occurs.

Software Note

The IEEE standard precludes saving and restoring the FPCR across subroutine calls.

IEEE Standard

The IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985) is included by reference.

• Memory Format Floating-Point Instructions

The instructions in this section move data between the floating-point registers and memory. They use the Memory instruction format. They do not interpret the bits moved in any way; specifically, they do not trap on non-finite values.

The instructions are summarized in Table 4-9.

Table 4-9 • Memory Format Floating-Point Instructions Summary

Mnemonic	Operation	Subset
LDF	Load F_floating	VAX
LDG	Load G_floating (Load D_floating)	VAX
LDS	Load S_floating (Load Longword Integer)	Both
LDT	Load T_floating (Load Quadword Integer)	Both
STF	Store F_floating	VAX
STG	Store G_floating (Store D_floating)	VAX
STS	Store S_floating (Store Longword Integer)	Both
STT	Store T_floating (Store Quadword Integer)	Both

Load F_floating

Format:

```
LDF     Fa.wf,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ←  (Rbv + SEXT(disp))

Fa ←  (va)<15> || MAP_F((va)<14:7>) ||
      (va)<6:0> || (va)<31:16> || 0<28:0>
```

Exceptions:

Access Violation

Fault on Read

Alignment

Translation Not Valid

Instruction mnemonics:

LDF Load F_floating

Qualifiers:

None

Description:

LDF fetches an F_floating datum from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The 8-bit memory-format exponent is expanded to an 11-bit register-format exponent according to Table 2-1.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory and the bytes are reordered to conform to the F_floating register format. The result is then zero-extended in the low-order longword and written to register Fa.

Load G_floating

Format:

```
LDG     Fa.wq,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ←   (Rbv + SEXT(disp))

Fa ←   (va)<15:0>  ||  (va)<31:16> ||
       (va)<47:32> ||  (va)<63:48>
```

Exceptions:

Access Violation

Fault on Read

Alignment

Translation Not Valid

Instruction mnemonics:

LDG Load G_floating (Load D_floating)

Qualifiers:

None

Description:

LDG fetches a G_floating (or D_floating) datum from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory, the bytes are reordered to conform to the G_floating register format (also conforming to the D_floating register format), and the result is then written to register Fa.

Load S_floating

Format:

```
LDS     Fa.ws,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ←   (Rbv + SEXT(disp))

Fa ←   (va)<31>     || MAP_S((va)<30:23>) ||
       (va)<22:0>   || 0<28:0>
```

Exceptions:

Access Violation

Fault on Read

Alignment

Translation Not Valid

Instruction mnemonics:

LDS	Load S_floating (Load Longword Integer)

Qualifiers:

None

Description:

LDS fetches a longword (integer or S_floating) from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The 8-bit memory-format exponent is expanded to an 11-bit register-format exponent according to Table 2-2.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory, is zero-extended in the low-order longword, and then written to register Fa.

Notes:

- Longword integers in floating registers are stored in bits <63:62,58:29>, with bits <61:59> ignored and zeros in bits <28:0>.

Load T_floating

Format:

```
LDT     Fa.wl,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ←   (Rbv + SEXT(disp))

Fa ←   (va)<63:0>
```

Exceptions:

Access Violation

Fault on Read

Alignment

Translation Not Valid

Instruction mnemonics:

LDT	Load T_floating (Load Quadword Integer)

Qualifiers:

None

Description:

LDT fetches a quadword (integer or T_floating) from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory and written to register Fa.

Store F_floating

Format:

```
STF     Fa.rf,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ← {Rbv + SEXT(disp)}

(va)<31:0> ← Fav<44:29> || Fav<63:62>|| Fav<58:45>
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STF Store F_floating

Qualifiers:

None

Description:

STF stores an F_floating datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The bits of the source operand are fetched from register Fa, the bits are reordered to conform to F_floating memory format, and the result is then written to memory. Bits <61:59> and <28:0> of Fa are ignored. No checking is done.

Store G_floating

Format:

```
STG     Fa.rg,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ← (Rbv + SEXT(disp));

(va)<63:0> ← Fav<15:0> || Fav<31:16> ||
             Fav<47:32> || Fav<63:48>
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STG	Store G_floating (Store D_floating)

Qualifiers:

None

Description:

STG stores a G_floating (or D_floating) datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from register Fa, the bytes are reordered to conform to the G_floating memory format (also conforming to the D_floating memory format), and the result is then written to memory.

Store S_floating

Format:

```
STS     Fa.rs,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ← (Rbv + SEXT(disp))

(va)<31:0> ← Fav<63:62>||Fav<58:29>
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STS	Store S_floating (Store Longword Integer)

Qualifiers:

None

Description:

STS stores a longword (integer or S_floating) datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The bits of the source operand are fetched from register Fa, the bits are reordered to conform to S_floating memory format, and the result is then written to memory. Bits <61:59> and <28:0> of Fa are ignored. No checking is done.

Store T_floating

Format:

```
STT     Fa.rt,disp.ab(Rb.ab)     !Memory format
```

Operation:

```
va ← (Rbv + SEXT(disp))

(va)<63:0> ← Fav<63:0>
```

Exceptions:

Access Violation

Fault on Write

Alignment

Translation Not Valid

Instruction mnemonics:

STT	Store T_floating (Store Quadword Integer)

Qualifiers:

None

Description:

STT stores a quadword (integer or T_floating) datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from register Fa and written to memory.

• Branch Format Floating-Point Instructions

Alpha provides six floating conditional branch instructions. These branch-format instructions test the value of a floating-point register and conditionally change the PC.

They do not interpret the bits tested in any way; specifically, they do not trap on non-finite values.

The test is based on the sign bit and whether the rest of the register is all zero bits. All 64 bits of the register are tested. The test is independent of the format of the operand in the register. Both plus and minus zero are equal to zero. A non-zero value with a sign of zero is greater than zero. A non-zero value with a sign of one is less than zero. No reserved operand or non-finite checking is done.

The floating-point branch operations are summarized in Table 4-10.

Table 4-10 • Floating-Point Branch Instructions Summary

Mnemonic	Operation	Subset
FBEQ	Floating Branch if Equal	Both
FBGE	Floating Branch if Greater Than or Equal	Both
FBGT	Floating Branch if Greater Than	Both
FBLE	Floating Branch if Less Than or Equal	Both
FBLT	Floating Branch if Less Than	Both
FBNE	Floating Branch if Not Equal	Both

Conditional Branch

Format:

```
FBxx    Fa.rq,disp.al                          !Branch format
```

Operation:

```
{update PC}
va ←  PC + {4*SEXT(disp)}
IF  TEST(Fav, Condition_based_on_Opcode)  THEN
    PC ←  va
```

Exceptions:

None

Instruction mnemonics:

FBEQ	Floating Branch if Equal
FBGE	Floating Branch if Greater Than or Equal
FBGT	Floating Branch if Greater Than
FBLE	Floating Branch if Less Than or Equal
FBLT	Floating Branch if Less Than
FBNE	Floating Branch if Not Equal

Qualifiers:

None

Description:

Register Fa is tested. If the specified relationship is true, the PC is loaded with the target virtual address; otherwise, execution continues with the next sequential instruction.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The conditional branch instructions are PC-relative only. The 21-bit signed displacement gives a forward/backward branch distance of +/– 1M instructions.

Notes:

- To branch properly on non-finite operands, compare to F31, then branch on the result of the compare.
- The largest negative integer ($8000\ 0000\ 0000\ 0000_{16}$) is the same bit pattern as floating minus zero, so it is treated as equal to zero by the branch instructions. To branch properly on the largest negative integer, convert it to floating or move it to an integer register and do an integer branch.

• Floating-Point Operate Format Instructions

The floating-point bit-operate instructions perform copy and integer convert operations on 64-bit register values. The bit-operate instructions do not interpret the bits moved in any way; specifically, they do not trap on non-finite values.

The floating-point arithmetic-operate instructions perform add, subtract, multiply, divide, compare, and floating convert operations on 64-bit register values in one of the four specified floating formats.

Each instruction specifies the source and destination formats of the values, as well as the rounding mode and trapping mode to be used. These instructions use the Floating-point Operate format.

The floating-point operate instructions are summarized in Table 4-11.

Table 4-11 • Floating-Point Operate Instructions Summary

Mnemonic	Operation	Subset
Bit and FPCR Operations		
CPYS	Copy Sign	Both
CPYSE	Copy Sign and Exponent	Both
CPYSN	Copy Sign Negate	Both
CVTLQ	Convert Longword to Quadword	Both
CVTQL	Convert Quadword to Longword	Both
FCMOVxx	Floating Conditional Move	Both
MF_FPCR	Move from Floating-point Control Register	Both
MT_FPCR	Move to Floating-point Control Register	Both

Table 4-11 • Floating-Point Operate Instructions Summary (Continued)

Mnemonic	Operation	Subset
Arithmetic Operations		
ADDF	Add F_floating	VAX
ADDG	Add G_floating	VAX
ADDS	Add S_floating	IEEE
ADDT	Add T_floating	IEEE
CMPGxx	Compare G_floating	VAX
CMPTxx	Compare T_floating	IEEE
CVTDG	Convert D_floating to G_floating	VAX
CVTGD	Convert G_floating to D_floating	VAX
CVTGF	Convert G_floating to F_floating	VAX
CVTGQ	Convert G_floating to Quadword	VAX
CVTQF	Convert Quadword to F_floating	VAX
CVTQG	Convert Quadword to G_floating	VAX
CVTQS	Convert Quadword to S_floating	IEEE
CVTQT	Convert Quadword to T_floating	IEEE
CVTTQ	Convert T_floating to Quadword	IEEE
CVTTS	Convert T_floating to S_floating	IEEE
DIVF	Divide F_floating	VAX
DIVG	Divide G_floating	VAX
DIVS	Divide S_floating	IEEE
DIVT	Divide T_floating	IEEE
MULF	Multiply F_floating	VAX
MULG	Multiply G_floating	VAX
MULS	Multiply S_floating	IEEE
MULT	Multiply T_floating	IEEE
SUBF	Subtract F_floating	VAX
SUBG	Subtract G_floating	VAX
SUBS	Subtract S_floating	IEEE
SUBT	Subtract T_floating	IEEE

Copy Sign

Format:

```
CPYSy   Fa.rq,Fb.rq,Fc.wq       !Floating-point Operate format
```

Operation:

```
CASE
        CPYS:   Fc ←  Fav<63> || Fbv<62:0>
        CPYSN:  Fc ←  NOT(Fav<63>) || Fbv<62:0>
        CPYSE:  Fc ←  Fav<63:52> || Fbv<51:0>
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

CPYS	Copy Sign
CPYSE	Copy Sign and Exponent
CPYSN	Copy Sign Negate

Qualifiers:

None

Description:

For CPYS and CPYSN, the sign bit of Fa is fetched (and complemented in the case of CPYSN) and concatenated with the exponent and fraction bits from Fb; the result is stored in Fc.

For CPYSE, the sign and exponent bits from Fa are fetched and concatenated with the fraction bits from Fb; the result is stored in Fc.

No checking of the operands is performed.

Notes:

- Register moves can be performed using CPYS Fx,Fx,Fy . Floating-point absolute value can be done using CPYS F31,Fx,Fy . Floating-point negation can be done using CPYSN Fx,Fx,Fy . Floating values can be scaled to a known range by using CPYSE.

Convert Integer to Integer

Format:

```
CVTxy      Fb.rq,Fc.wx          !Floating-point Operate format
```

Operation:

```
CASE
     CVTQL: Fc ← Fbv<31:30> || 0<2:0> ||
                 Fbv<29:0>  || 0<28:0>

     CVTLQ: Fc ← SEXT(Fbv<63:62> || Fbv<58:29>)
ENDCASE
```

Exceptions:

Integer Overflow, CVTQL only

Instruction mnemonics:

CVTLQ	Convert Longword to Quadword
CVTQL	Convert Quadword to Longword

Qualifiers:

Trapping:	Software (/S)
	Integer Overflow Enable (/V) (CVTQL only)

Description:

The two's-complement operand in register Fb is converted to a two's-complement result and written to register Fc.

The conversion from quadword to longword is a repositioning of the low 32 bits of the operand, with zero fill and optional integer overflow checking. Integer overflow occurs if Fb is outside the range –2**31..2**31–1. If integer overflow occurs, the truncated result is stored in Fc, and an arithmetic trap is taken if enabled.

The conversion from longword to quadword is a repositioning of 32 bits of the operand, with sign extension.

Floating-Point Conditional Move

Format:

```
FCMOVxx Fa.rq,Fb.rq,Fc.wq        !Floating-point Operate format
```

Operation:

```
IF  TEST(Fav, Condition_based_on_Opcode)  THEN

    Fc ←  Fbv
```

Exceptions:

None

Instruction mnemonics:

FCMOVEQ	FCMOVE if Register Equal to Zero
FCMOVGE	FCMOVE if Register Greater Than or Equal to Zero
FCMOVGT	FCMOVE if Register Greater Than Zero
FCMOVLE	FCMOVE if Register Less Than or Equal to Zero
FCMOVLT	FCMOVE if Register Less Than Zero
FCMOVNE	FCMOVE if Register Not Equal to Zero

Qualifiers:

None

Description:

Register Fa is tested. If the specified relationship is true, register Fb is written to register Fc; otherwise, the move is suppressed and register Fc is unchanged. The test is based on the sign bit and whether the rest of the register is all zero bits, as described for floating branches in *Branch Format Floating-Point Instructions* in this chapter.

Notes:

Except that it is likely in many implementations to be substantially faster, the instruction:

```
FCMOVxx Fa,Fb,Fc
```

is exactly equivalent to:

```
        FByy Fa,label    ; yy = NOT xx
        CPYS Fb,Fb,Fc
label:  ...
```

For example, a branchless sequence for:

```
F1=MAX(F1,F2)
```

is:

```
        CMPxLT  F1,F2,F3        ! F3=one if F1<F2; x=F/G/S/T
        FCMOVNE F3,F2,F1        ! Move F2 to F1 if F1<F2
```

Move from/to Floating-Point Control Register

Format:

```
Mx_FPCR Fa.rq,Fa.rq,Fa.wq        !Floating-point Operate format
```

Operation:

```
CASE
   MT_FPCR:  FPCR ← Fav
   MF_FPCR:  Fa   ← FPCR
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

MF_FPCR	Move from Floating-point Control Register
MT_FPCR	Move to Floating-point Control Register

Qualifiers:

None

Description:

The Floating-point Control Register (FPCR) is read from (MF_FPCR) or written to (MT_FPCR), a floating-point register. The floating-point register to be used is specified by the Fa, Fb, and Fc fields all pointing to the same floating-point register. If the Fa, Fb, and Fc fields do not all point to the same floating-point register, then it is UNPREDICTABLE which register is used.

The use of these instructions and the FPCR are described in *FPCR Register and Dynamic Rounding Mode* in this chapter.

VAX Floating Add

Format:

```
ADDx     Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ← Fav + Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Instruction mnemonics:

ADDF	Add F_floating
ADDG	Add G_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U)

Description:

Register Fa is added to register Fb, and the sum is written to register Fc.

The sum is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow or underflow.

IEEE Floating Add

Format:

```
ADDx     Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ←  Fav + Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Inexact Result

Instruction mnemonics:

ADDS	Add S_floating
ADDT	Add T_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U) Inexact Enable (/I)

Description:

Register Fa is added to register Fb, and the sum is written to register Fc.

The sum is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow, underflow, or inexact result.

VAX Floating Compare

Format:

```
CMPGyy   Fa.rg,Fb.rg,Fc.wq          !Floating-point Operate format
```

Operation:

```
IF   Fav SIGNED_RELATION Fbv  THEN
     Fc ←   4000 0000 0000 0000₁₆
ELSE
     Fc ←   0000 0000 0000 0000₁₆
```

Exceptions:

Invalid Operation

Instruction mnemonics:

CMPGEQ	Compare G_floating Equal
CMPGLE	Compare G_floating Less Than or Equal
CMPGLT	Compare G_floating Less Than

Qualifiers:

Trapping:	Software (/S)

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (0.5) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Three mutually exclusive relations are possible: less than, equal, and greater than.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

Notes:

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.

IEEE Floating Compare

Format:

```
CMPTyy  Fa.rx,Fb.rx,Fc.wq        !Floating-point Operate format
```

Operation:

```
IF  Fav SIGNED_RELATION Fbv  THEN
    Fc ←  4000 0000 0000 0000₁₆
ELSE
    Fc ←  0000 0000 0000 0000₁₆
```

Exceptions:

Invalid Operation

Instruction mnemonics:

CMPTEQ	Compare T_floating Equal
CMPTLE	Compare T_floating Less Than or Equal
CMPTLT	Compare T_floating Less Than
CMPTUN	Compare T_floating Unordered

Qualifiers:

Trapping:	Software (/S)

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (2.0) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Four mutually exclusive relations are possible: less than, equal, greater than, and unordered. The unordered relation is true if one or both operands are NaN. (This behavior must be provided by a software trap handler, since NaNs trap.) Comparisons ignore the sign of zero, so +0 = −0 .

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones and a non-zero fraction (IEEE NaNs). The contents of Fc are UNPREDICTABLE if this occurs.

Comparisons with plus and minus infinity execute normally and do not take an invalid operation trap.

Notes:

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.

Convert VAX Floating to Integer

Format:

```
CVTGQ      Fb.rx,Fc.wq              !Floating-point Operate format
```

Operation:

```
Fc ← (conversion of Fbv)
```

Exceptions:

Invalid Operation

Integer Overflow

Instruction mnemonics:

CVTGQ	Convert G_floating to Quadword

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Integer Overflow Enable (/V)

Description:

The floating operand in register Fb is converted to a two's-complement quadword number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on integer overflow.

Convert Integer to VAX Floating

Format:

```
CVTQy      Fb.rq,Fc.wx              !Floating-point Operate format
```

Operation:

```
Fc ←  {conversion of Fbv<63:0>}
```

Exceptions:

None

Instruction mnemonics:

CVTQF	Convert Quadword to F_floating
CVTQG	Convert Quadword to G_floating

Qualifiers:

Rounding:	Chopped (/C)

Description:

The two's-complement quadword operand in register Fb is converted to a single- or double-precision floating result and written to register Fc. The conversion complements a number if negative, normalizes it, rounds to the target precision, and packs the result with an appropriate sign and exponent field.

Convert VAX Floating to VAX Floating

Format:

```
CVTxy      Fb.rx,Fc.wx          !Floating-point Operate format
```

Operation:

```
Fc ← (conversion of Fbv)
```

Exceptions:

Invalid Operation

Overflow

Underflow

Instruction mnemonics:

CVTDG	Convert D_floating to G_floating
CVTGD	Convert G_floating to D_floating
CVTGF	Convert G_floating to F_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U)

Description:

The floating operand in register Fb is converted to the specified alternate floating format and written to register Fc.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow or underflow.

Notes:

- The only arithmetic operations on D_floating values are conversions to and from G_floating. The conversion to G_floating rounds or chops as specified, removing three fraction bits. The conversion from G_floating to D_floating adds three low-order zeros as fraction bits, then the 8-bit exponent range is checked for overflow/underflow.
- The conversion from G_floating to F_floating rounds or chops to single precision, then the 8-bit exponent range is checked for overflow/underflow.
- No conversion from F_floating to G_floating is required, since F_floating values are always stored in registers as equivalent G_floating values.

Convert IEEE Floating to Integer

Format:

```
CVTTQ      Fb.rx,Fc.wq          !Floating-point Operate format
```

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation

Inexact Result

Integer Overflow

Instruction mnemonics:

CVTTQ	Convert T_floating to Quadword

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Integer Overflow Enable (/V) Inexact Enable (/I)

Description:

The floating operand in register Fb is converted to a two's-complement number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on integer overflow and inexact result.

Convert Integer to IEEE Floating

Format:

```
CVTQy      Fb.rq,Fc.wx            !Floating-point Operate format
```

Operation:

```
Fc ←  {conversion of Fbv<63:0>}
```

Exceptions:

Inexact Result

Instruction mnemonics:

CVTQS	Convert Quadword to S_floating
CVTQT	Convert Quadword to T_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Inexact Enable (/I)

Description:

The two's-complement operand in register Fb is converted to a single- or double-precision floating result and written to register Fc. The conversion complements a number if negative, normalizes it, rounds to the target precision, and packs the result with an appropriate sign and exponent field.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on inexact result.

Convert IEEE Floating to IEEE Floating

Format:

```
CVTTS     Fb.rx,Fc.wx          !Floating-point Operate format
```

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation

Overflow

Underflow

Inexact Result

Instruction mnemonics:

CVTTS	Convert T_floating to S_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U) Inexact Enable (/I)

Description:

The floating operand in register Fb is converted to the specified alternate floating format and written to register Fc.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow, underflow, or inexact result.

Notes:

- No conversion from S_floating to T_floating is required, since S_floating values are always stored in registers as equivalent T_floating values.

VAX Floating Divide

Format:

```
DIVx     Fa.rx,Fb.rx,Fc.wx          !Floating-point Operate format
```

Operation:

```
Fc ← Fav / Fbv
```

Exceptions:

Invalid Operation

Division by Zero

Overflow

Underflow

Instruction mnemonics:

DIVF	Divide F_floating
DIVG	Divide G_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U)

Description:

The dividend operand in register Fa is divided by the divisor operand in register Fb, and the quotient is written to register Fc.

The quotient is rounded or chopped to the specified precision and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

A division by zero trap is signaled if Fbv is zero. The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow or underflow.

IEEE Floating Divide

Format:

```
DIVx    Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ← Fav / Fbv
```

Exceptions:

Invalid Operation

Division by Zero

Overflow

Underflow

Inexact Result

Instruction mnemonics:

DIVS	Divide S_floating
DIVT	Divide T_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U) Inexact Enable (/I)

Description:

The dividend operand in register Fa is divided by the divisor operand in register Fb, and the quotient is written to register Fc.

The quotient is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

A division by zero trap is signaled if Fbv is zero. The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow, underflow, or inexact result.

VAX Floating Multiply

Format:

```
MULx     Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ← Fav * Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Instruction mnemonics:

MULF	Multiply F_floating
MULG	Multiply G_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow or underflow.

IEEE Floating Multiply

Format:

```
MULx     Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ←  Fav * Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Inexact Result

Instruction mnemonics:

MULS	Multiply S_floating
MULT	Multiply T_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Underflow Eenable (/U) Inexact Enable (/I)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow, underflow, or inexact result.

VAX Floating Subtract

Format:

```
SUBx     Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ← Fav - Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Instruction mnemonics:

SUBF	Subtract F_floating
SUBG	Subtract G_floating

Qualifiers:

Rounding:	Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands *and* dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow or underflow.

IEEE Floating Subtract

Format:

```
SUBx    Fa.rx,Fb.rx,Fc.wx        !Floating-point Operate format
```

Operation:

```
Fc ← Fav - Fbv
```

Exceptions:

Invalid Operation

Overflow

Underflow

Inexact Result

Instruction mnemonics:

SUBS	Subtract S_floating
SUBT	Subtract T_floating

Qualifiers:

Rounding:	Dynamic (/D) Minus infinity (/M) Chopped (/C)
Trapping:	Software (/S) Underflow Enable (/U) Inexact Enable (/I)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and a non-zero fraction (IEEE denormals trap), or if exp=all-ones (IEEE NaNs and infinities trap).

The contents of Fc are UNPREDICTABLE if this occurs.

See *Floating-Point Trapping Modes* in this chapter for details of the stored result on overflow, underflow, or inexact result.

• Miscellaneous Instructions

Alpha provides the miscellaneous instructions shown in Table 4-12.

Table 4-12 • Miscellaneous Instructions Summary

Mnemonic	Operation
CALL_PAL	Call Privileged Architecture Library Routine
FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent
MB	Memory Barrier
RPCC	Read Process Cycle Counter
TRAPB	Trap Barrier

Call Privileged Architecture Library

Format:

```
CALL_PAL fnc.ir                    !PAL format
```

Operation:

```
{Stall instruction issuing until all
prior instructions are guaranteed to
complete without incurring exceptions.}
{Trap to PAL code.}
```

Exceptions:

None

Instruction mnemonics:

CALL_PAL Call Privileged Architecture Library

Qualifiers:

None

Description:

The CALL_PAL instruction is not issued until all previous instructions are guaranteed to complete without exceptions. If an exception occurs, the continuation PC in the exception stack frame points to the CALL_PAL instruction. The CALL_PAL instruction causes a trap to PAL code.

Prefetch Data

Format:

```
FETCHx   0(Rb.ab)                    !Memory format
```

Operation:

```
va ← (Rbv)
(Optionally prefetch aligned 512-byte block surrounding va.)
```

Exceptions:

None

Instruction mnemonics:

FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent

Qualifiers:

None

Description:

The virtual address is given by Rbv. This address is used to designate an aligned 512-byte block of data. An implementation may optionally attempt to move all or part of this block (or a larger surrounding block) of data to a faster-access part of the memory hierarchy, in anticipation of subsequent Load or Store instructions that access that data.

The FETCH instruction is a hint to the implementation that may allow faster execution. An implementation is free to ignore the hint. If prefetching is done in an implementation, the order of fetch within the designated block is UNPREDICTABLE.

The FETCH_M instruction gives the additional hint that modifications (stores) to some or all of the data block are anticipated.

No exceptions are generated by FETCHx. If a Load (or Store in the case of FETCH_M) that uses the same address would fault, the prefetch request is ignored. It is UNPREDICTABLE whether a TB-miss fault is ever taken by FETCHx.

Implementation Note

Implementations are encouraged to take the TB-miss fault, then continue the prefetch.

The programming model for effective use of FETCH and FETCH_M is given in Appendix A.

Software Note

FETCH is intended to help software overlap memory latencies on the order of 100 cycles. FETCH is unlikely to help (or be implemented) for memory latencies on the order of 10 cycles. Code scheduling should be used to overlap such short latencies.

Memory Barrier

Format:

```
MB                                    !Memory format
```

Operation:

```
{Guarantee that all subsequent loads or stores
will not access memory until after all previous
loads and stores have accessed memory, as
observed by other processors.}
```

Exceptions:

None

Instruction mnemonics:

MB	Memory Barrier

Qualifiers:

None

Description:

The use of the Memory Barrier (MB) instruction is required only in multiprocessor systems.

In the absence of an MB instruction, loads and stores to different physical locations are allowed to complete out of order on the issuing processor as observed by other processors. The MB instruction allows memory accesses to be serialized on the issuing processor as observed by other processors. See Chapter 5 for details on using the MB instruction to serialize these accesses. Chapter 5 also details coordinating memory accesses across processors.

Note that MB ensures serialization only; it does not necessarily accelerate the progress of memory operations.

Read Process Cycle Counter

Format:

```
RPCC      Ra.wq              !Memory format
```

Operation:

```
Ra ← {cycle counter}
```

Exceptions:

None

Instruction mnemonics:

RPCC	Read Process Cycle Counter

Qualifiers:

None

Description:

Register Ra is written with the process cycle counter (PCC).

The low-order 32 bits of the process cycle counter is an unsigned 32-bit integer that increments once per N CPU cycles, where N is an implementation-specific integer in the range 1..16. The cycle counter frequency is the number of times the process cycle counter gets incremented per second, rounded to a 64-bit integer. The integer count wraps to 0 from a count of FFFF FFFF$_{16}$. The counter wraps no more frequently than 1.5 times the implementation's interval clock interrupt period (which is two thirds of the interval clock interrupt frequency). The high-order 32 bits of the process cycle counter are an offset that when added to the low-order 32 bits gives the cycle count for this process.

The process cycle counter is suitable for timing intervals on the order of nanoseconds and may be used for detailed performance characterization. It is required on all implementations. PCC is required for every processor, and each processor in a multiprocessor system has its own private, independent PCC.

As an example, consider the following code that returns in R0 the current cycle count MOD 2**32.

```
RPCC    R0            ; Read the process cycle counter
SLL     R0, #32, R1   ; line up the offset and count fields
ADDQ    R0, R1, R0    ; do add
SRL     R0, #32, R0   ; zero extend the cycle count to 64 bits
```

Trap Barrier

Format:

```
TRAPB                                   !Memory format
```

Operation:

```
{Stall instruction issuing until all prior instructions are
guaranteed to complete without incurring arithmetic traps.}
```

Exceptions:

None

Instruction mnemonics:

TRAPB Trap Barrier

Qualifiers:

None

Description:

The TRAPB instruction allows software to guarantee that in a pipelined implementation, all previous arithmetic instructions will complete without incurring any arithmetic traps before any instructions after the TRAPB are issued. For example, TRAPB should be used before changing an exception handler to ensure that all exceptions on previous instructions are processed in the current exception-handling environment.

▪ VAX Compatibility Instructions

Alpha provides the instructions shown in Table 4-13 for use in translated VAX code. These instructions are not a permanent part of the architecture and will not be available in some future implementations. They are intended to preserve customer assumptions about VAX instruction atomicity in porting code from VAX to Alpha.

These instructions should be generated only by the VAX-to-Alpha software translator; they should never be used in native Alpha code. Any native code that uses them may cease to work.

Table 4-13 • VAX Compatibility Instructions Summary

Mnemonic	Operation
RC	Read and Clear
RS	Read and Set

VAX Compatibility Instructions

Format:

```
Rx      Ra.wq           !Memory format
```

Operation:

```
Ra ←  intr_flag
intr_flag ←  0          !RC
intr_flag ←  1          !RS
```

Exceptions:

None

Instruction mnemonics:

RC	Read and Clear
RS	Read and Set

Qualifiers:

None

Description:

The intr_flag is returned in Ra and then cleared to zero (RC) or set to one (RS).

These instructions may be used to determine whether the sequence of Alpha instructions between RS and RC (corresponding to a single VAX instruction) was executed without interruption or exception.

Intr_flag is a per-processor state bit. The intr_flag is cleared if that processor encounters any exception or interrupt.

It is UNPREDICTABLE whether a processor's intr_flag is affected when that processor executes an LDx_L or STx_C instruction. A processor's intr_flag is not affected when that processor executes a normal load or store instruction.

A processor's intr_flag is not affected when that processor executes a taken branch.

Note

These instructions are intended *only* for use by the VAX-to-Alpha software translator; they should never be used by native code.

Chapter 5 • System Architecture and Programming Implications

• Introduction

Portions of the Alpha architecture have implications for programming, and the system structure, of both uniprocessor and multiprocessor implementations. Architectural implications considered in the following sections are:

- Physical memory behavior
- Caches and write buffers
- Translation buffers and virtual caches
- Data sharing
- Read/write ordering
- Stacks
- Arithmetic traps

To meet the requirements of the Alpha architecture, software and hardware implementors need to take these issues into consideration.

• Physical Memory Behavior

Alpha physical memory space is divided into four regions, based on the two most significant, implemented, physical address bits. Each region's behavior can be described in terms of its coherency, granularity, width, and memory-like behavior.

Coherency of Memory Access

Alpha implementations must provide a coherent view of memory, in which each write by a processor or I/O device (hereafter, called "processor") becomes visible to all other processors. No distinction is made between coherency of "memory space" and "I/O space".

Memory coherency may be provided in different ways, for each of the four physical address regions.

Possible per-region policies include, but are not restricted to:

1. No caching

   No copies are kept of data in a region; all reads and writes access the actual data location (memory or I/O register).

2. Write-through caching

   Copies are kept of any data in the region; reads may use the copies, but writes update the actual data location and either update or invalidate all copies.

3. Write-back caching

   Copies are kept of any data in the region; reads and writes may use the copies, and writes use additional state to determine whether there are other copies to invalidate or update.

Part of the coherency policy implemented for a given physical address region may include restrictions on excess data transfers (performing more accesses to a location than is necessary to acquire or change the location's value), or may specify data transfer widths (the granularity used to access a location).

Independent of coherency policy, a processor may use different hardware or different hardware resource policies for caching or buffering different physical address regions.

Granularity of Memory Access

For each region, an implementation must support aligned quadword access and may optionally support aligned longword access.

For a quadword access region, accesses to physical memory must be implemented such that independent accesses to adjacent aligned quadwords produce the same results regardless of the order of execution. Further, an access to an aligned quadword must be done in a single atomic operation.

For a longword access region, accesses to physical memory must be implemented such that independent accesses to adjacent aligned longwords produce the same results regardless of the order of execution. Further, an access to an aligned longword must be done in a single atomic operation, and an access to an aligned quadword must also be done in a single atomic operation.

In this context, "atomic" means that if different processors do simultaneous reads and writes of the same data, it must not be possible to observe a partial write of the subject longword or quadword.

Width of Memory Access

Subject to the granularity, ordering, and coherency constraints given in the sections of this chapter entitled *Coherency of Memory Access*, *Granularity of Memory Access*, and *Read/Write Ordering*, accesses to physical memory may be freely cached, buffered, and prefetched.

A processor may read more physical memory data (such as a full cache block) than is actually accessed, writes may trigger reads, and writes may write back more data than is actually updated. A processor may elide multiple reads and/or writes to the same data.

Memory-Like Behavior

A memory-like region obeys the following rules:

- Each page frame in the region either exists in its entirety or does not exist in its entirety; there are no holes within a page frame.
- All locations that exist are read/write.
- A write to a location followed by a read from that location returns precisely the bits written; all bits act as memory.
- A write to one location does not change any other location.
- Reads have no side effects.
- Longword access granularity is provided.
- Instruction-fetch is supported.
- Load-locked and store-conditional are supported.

Non-memory-like regions may have much more arbitrary behavior:

- Unimplemented locations or bits may exist anywhere.
- Some locations or bits may be read-only and others write-only.
- Address ranges may overlap, such that a write to one location changes the bits read from a different location.
- Reads may have side effects, although this is strongly discouraged.
- Longword granularity need not be supported.
- Instruction-fetch need not be supported.
- Load-locked and store-conditional need not be supported.

Hardware/Software Coordination Note

The details of such behavior are outside the scope of the Alpha architecture. Specific processor and I/O adapter implementations may choose and document whatever behavior they need. It is the responsibility of system designers to impose enough consistency to allow processors successfully to access matching non-memory devices in a coherent way.

Translation Buffers and Virtual Caches

A system may choose to include a Translation Buffer (TB), a virtual instruction cache (virtual I-cache), or a virtual data cache (virtual D-cache). The contents of these caches and/or translation buffers may become invalid, depending upon what operating system activity is being performed.

Whenever a nonsoftware field of a valid Page Table Entry (PTE) is modified, copies of that PTE must be made coherent. Translation Buffer (TB) entries and virtual D-cache entries can be made coherent by calling the appropriate PALcode routine to invalidate the TB. Virtual I-cache entries can be made coherent via the IMB PAL call.

If a processor implements address space numbers (ASNs), and the old PTE has the address space match (ASM) bit clear (ASNs in use) and the valid bit set, then entries can also effectively be made coherent by assigning a new, unused ASN to the currently running process and not reusing the previous ASN before calling the appropriate PALcode routine to invalidate the Translation Buffer (TB).

In a multiprocessor environment, making the TBs and/or caches coherent on only one processor is not always sufficient. An operating system must arrange to perform the above actions on each processor that could possibly have copies of the PTE or data for any affected page.

Caches and Write Buffers

A hardware implementation may include mechanisms to reduce memory access time by making local copies of recently used memory contents (or those expected to be used) or by buffering writes to complete at a later time. Caches and write buffers are examples of these mechanisms. They must be implemented so that their existence is transparent to software (except for timing, error reporting/control/recovery, and modification to the I-stream).

The following requirements must be met by all cache/write-buffer implementations. All processors must provide a coherent view of memory.

1. Write buffers may be used to delay and aggregate writes. From the viewpoint of another processor, buffered writes appear not to have happened yet. (Write buffers must not delay writes indefinitely. See *Timeliness*.)
2. Write-back caches must be able to detect a later write from another processor and invalidate or update the cache contents.
3. A processor must guarantee that a data store to a location followed by a data load from the same location must read the updated value.
4. Cache prefetching is allowed, but virtual caches must not prefetch from invalid pages.
5. A processor must guarantee that all of its previous writes are visible to all other processors before a HALT instruction completes. A processor must guarantee that its caches are coherent with the rest of the system before continuing from a HALT.
6. If battery backup is supplied, a processor must guarantee that the memory system remains coherent across a powerfail/recovery sequence. Data that was written by the processor before the powerfail may not be lost, and any caches must be in a valid state before (and if) normal instruction processing is continued after power is restored.
7. Virtual instruction caches are not required to notice modifications of the virtual I-stream (they need not be coherent with the rest of memory). Software that creates or modifies the instruction stream must execute an IMB PAL call before trying to execute the new instructions.

   For example, if two different virtual addresses, VA1 and VA2, map to the same page frame, a store to VA1 modifies the virtual I-stream fetched via VA2.

However, the sequence:

–Change the mapping of an I-stream page from valid to invalid, then

–Copy the corresponding page frame to a new page frame, then

–Change the original mapping to be valid and point to the new page frame

does not modify the virtual I-stream (this might happen in soft page faults).

8. Physical instruction caches are not required to notice modifications of the physical I-stream (they need not be coherent with the rest of memory), except for certain paging activity. (See *Timeliness*.) Software that creates or modifies the instruction stream must execute an IMB PAL call before trying to execute the new instructions.

   In this context, to "modify the physical I-stream" means any Store to the same physical address that is subsequently fetched as an instruction.

In this context, to "modify the virtual I-stream" means any Store to the same physical address that is subsequently fetched as an instruction via some corresponding (virtual address, ASN) pair, or to change the virtual-to-physical address mapping so that different values are fetched.

• Data Sharing

In a multiprocessor environment, writes to shared data must be synchronized by the programmer.

Atomic Change of a Single Datum

The ordinary STL and STQ instructions can be used to perform an atomic change of a shared aligned longword or quadword. ("Change" means that the new value is not a function of the old value.) In particular, an ordinary STL or STQ instruction can be used to change a variable that could be simultaneously accessed via an LDx_L/STx_C sequence.

Atomic Update of a Single Datum

The load-locked/store-conditional instructions may be used to perform an atomic update of a shared aligned longword or quadword. ("Update" means that the new value is a function of the old value.)

The following sequence performs a read-modify-write operation on location *x*. Only register-to-register operate instructions and branch fall-throughs may occur in the sequence:

```
try_again:
        LDQ_L   R1,x
        <modify R1>
        STQ_C   R1,x
        BEQ     R1,no_store
          :
          :
no_store:
        <code to check for excessive iterations>
        BR      try_again
```

If this sequence runs with no exceptions or interrupts, and no other processor writes to location *x* (more precisely, the locked range including *x*) between the LDQ_L and STQ_C instructions, then the STQ_C shown in the example stores the modified value in *x* and sets R1 to 1. If, however, the sequence encounters exceptions or interrupts that eventually continue the sequence, or another processor writes to *x*, then the STQ_C does not store and sets R1 to 0. In this case, the sequence is repeated via the branches to no_store and try_again. This repetition continues until the reasons for exceptions or interrupts are removed, and no interfering store is encountered.

To be useful, the sequence must be constructed so that it can be replayed an arbitrary number of times, giving the same result values each time. A sufficient (but not necessary) condition is that, within the sequence, the set of operand destinations and the set of operand sources are disjoint.

Note

A sufficiently long instruction sequence between LDQ_L and STQ_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The rules in Appendix A describe sequences that will eventually complete in *all* Alpha implementations.

This load-locked/store-conditional paradigm may be used whenever an atomic update of a shared aligned quadword is desired, including getting the effect of atomic byte writes.

Atomic Update of Data Structures

Before accessing shared writable data structures (those that are not a single aligned longword or quadword), the programmer can acquire control of the data structure by using an atomic update to set a software lock variable. Such a software lock can be cleared with an ordinary store instruction.

A software-critical section, therefore, may look like the sequence:

```
stq_c_loop:
spin_loop:
        LDQ_L R1,lock_variable     \
        BLBS  R1,already_set        \
        OR    R1,#1,R2               > Set lock bit
        STQ_C R2,lock_variable      /
        BEQ   R2,stq_c_fail        /

        MB
        <critical section: updates various data structures>
        MB

        STQ   R31,lock_variable      ; Clear lock bit
         :
         :
already_set:
        <code to block or reschedule or test for too many iterations>
        BR    spin_loop
stq_c_fail:
        <code to test for too many iterations>
        BR    stq_c_loop
```

This code has a number of subtleties:

1. If the lock_variable is already set, the spin loop is done without doing any stores. This avoidance of stores improves memory subsystem performance and avoids the deadlock described below.
2. If the lock_variable is actually being changed from 0 to 1, and the STQ_C fails (due to an interrupt, or because another processor simultaneously changed lock_variable), the entire process starts over by reading the lock_variable again.
3. Only the fall-through path of the BLBS does a STx_C; some implementations may not allow a successful STx_C after a branch-taken.
4. Only register-to-register operate instructions are used to do the modify.
5. Both conditional branches are forward branches, so they are properly predicted not to be taken (to match the common case of no contention for the lock).
6. The OR writes its result to a second register; this allows the OR and the BLBS to be interchanged if that would give a faster instruction schedule.
7. Other operate instructions (from the critical section) may be scheduled into the LDQ_L..STQ_C sequence, so long as they do not fault or trap, and they give correct results if repeated; other memory or operate instructions may be scheduled between the STQ_C and BEQ.
8. The MB instructions are discussed in *Ordering Considerations for Shared Data Structures.*
9. An ordinary STQ instruction is used to clear the lock_variable.

It would be a performance mistake to spin-wait by repeating the full LDQ_L..STQ_C sequence (to move the BLBS after the BEQ) because that sequence may repeatedly change the software lock_variable from "locked" to "locked," with each write causing extra access delays in all other caches that contain the lock_variable. In the extreme, spin-waits that contain writes may deadlock as follows:

> If, when one processor spins with writes, another processor is modifying (not changing) the lock_variable, then the writes on the first processor may cause the STx_C of the modify on the second processor always to fail.

This deadlock situation is avoided by:

- Having only one processor do a store (no STx_C), or
- Having no write in the spin loop, or
- Doing a write *only* if the shared variable actually changes state (1 → 1 does not change state).

Ordering Considerations for Shared Data Structures

A critical section sequence, such as shown in *Atomic Update of Data Structures*, is conceptually only three steps:

1. Acquire software lock
2. Critical section—read/write shared data
3. Clear software lock

In the absence of explicit instructions to the contrary, the Alpha architecture allows reads and writes to be reordered. While this may allow more implementation speed and overlap, it can also create undesired side effects on shared data structures. Normally, the critical section just described would have two instructions added to it:

```
<acquire software lock>
 MB (memory barrier #1)
<critical section -- read/write shared data>
 MB (memory barrier #2)
<clear software lock>
```

The first memory barrier prevents any reads (from within the critical section) from being prefetched before the software lock is acquired; such prefetched reads would potentially contain stale data.

The second memory barrier prevents any reads or writes (from within the critical section) from being delayed past the clearing of the software lock; such delayed accesses could interact with the next user of the shared data, defeating the purpose of the software lock entirely.

Software Note

In the VAX architecture, many instructions provide noninterruptable read-modify-write sequences to memory variables. Most programmers never regard data sharing as an issue.

In the Alpha architecture, programmers must pay more attention to synchronizing access to shared data; for example, to AST routines. In the VAX, a programmer can use an ADDL2 to update a variable that is shared between a "MAIN" routine and an AST routine, if running on a single processor. In the Alpha architecture, a programmer must deal with AST shared data by using multiprocessor shared data sequences.

• Read/Write Ordering

This section does not apply to programs that run on a single processor and do not write to the instruction stream. On a single processor, all memory accesses appear to happen in the order specified by the programmer. This section deals entirely with predictable read/write ordering across multiple processors.

The order of reads and writes done in an Alpha implementation may differ from that specified by the programmer.

For any two memory references A and B, either A must occur before B in all Alpha implementations, B must occur before A, or they are UNORDERED. In the last case, software cannot depend upon one occurring first: the order may vary from implementation to implementation, and even from run to run or moment to moment on a single implementation.

If two references cannot be shown to be ordered by the rules given, they are UNORDERED and implementations are free to do them in any order that is convenient. Implementations may take advantage of this freedom to deliver substantially higher performance.

The discussion that follows first defines the architectural issue sequence of memory references on a single processor, then defines the (partial) ordering on this issue sequence that *all* Alpha implementations are required to maintain.

The individual issue sequences on multiple processors are merged into access sequences at each shared memory location. The discussion defines the (partial) ordering on the individual access sequences that *all* Alpha implementations are required to maintain.

The net result is that for any code that executes on multiple processors, one can determine which memory accesses are required to occur before others on *all* Alpha implementations and hence can write useful shared-variable software.

Software writers can force one reference to occur before another by inserting a memory barrier instruction (MB or IMB) between the references.

Alpha Shared Memory Model

An Alpha system consists of a collection of *processors* and shared coherent *memories* that are accessible by all processors. (There may also be unshared memories, but they are outside the scope of this section.)

A *processor* is an Alpha CPU or an I/O device (or anything else that gets added).

A *shared memory* is the primary storage place for one or more locations.

A *location* is an aligned quadword, specified by its physical address. Multiple virtual addresses may map to the same physical address. Ordering considerations are based only on the physical address.

Implementation Note

An implementation may allow a location to have multiple physical addresses, but the rules for accesses via mixtures of the addresses are implementation-specific and outside the scope of this section. Accesses via exactly one of the physical addresses follow the rules described next.

Each processor may generate *accesses* to shared memory locations. There are five types of accesses:

1. Instruction fetch by processor *i* to location *x*, returning value *a*, denoted Pi:I(x,a) .
2. Data read by processor *i* to location *x*, returning value *a*, denoted Pi:R(x,a) .
3. Data write by processor *i* to location *x*, storing value *a*, denoted Pi:W(x,a) .
4. Memory barrier instruction issued by processor *i*, denoted Pi:MB .
5. I-stream memory barrier instruction issued by processor *i*, denoted Pi:IMB .

The first access type is also called an I-stream access or I-fetch. The next two are also called D-stream accesses. The first three types collectively are called read/write accesses, denoted Pi:*(x,a). The last two types collectively are called barriers.

During actual execution in an Alpha system, each processor has a time-ordered *issue sequence* of all the memory references presented by that processor (to all memory locations), and each location has a time-ordered *access sequence* of all the accesses presented to that location (from all processors).

Architectural Definition of Processor Issue Sequence

The issue sequence for a processor is architecturally defined with respect to a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers. This is the instruction execution model:

1. I-fetch: An Alpha instruction is fetched from memory.
2. Read/Write: That instruction is executed and runs to completion, including a single data read from memory for a Load instruction or a single data write to memory for a Store instruction.
3. Update: The PC for the processor is updated.
4. Loop: Repeat the above sequence indefinitely.

If the instruction fetch step gets a memory management fault, the I-fetch is not done and the PC is updated to point to a PALcode fault handler. If the read/write step gets a memory management fault, the read/write is not done and the PC is updated to point to a PALcode fault handler.

All memory references are aligned quadwords. For the purpose of defining ordering, aligned longword references are modeled as quadword references to the containing aligned quadword.

Definition of Processor Issue Order

A partial ordering, called processor issue order, is imposed on the issue sequence defined in *Architectural Definition of Processor Issue Sequence* in this chapter.

For two accesses u and v issued by processor Pi, u is said to PRECEDE v IN ISSUE ORDER (<) if u occurs earlier than v in the issue sequence for Pi, and either of the following applies:

1. The access types are of the following issue order:

Table 5-1 • Processor Issue Order

1st↓/2nd→	Pi:I(y,b)	Pi:R(y,b)	Pi:W(y,b)	Pi:MB	Pi:IMB
Pi:I(x,a)	< if x=y		< if x=y	<	<
Pi:R(x,a)		< if x=y	< if x=y	<	<
Pi:W(x,a)		< if x=y	< if x=y	<	<
Pi:MB		<	<	<	<
Pi:IMB	<	<	<	<	<

2. Or, u is a TB fill, for example, a PTE read in order to satisfy a TB miss, and v is an I- or D-stream access using that PTE (see *Litmus Tests*).

Issue order is thus a partial order imposed on the architecturally specified issue sequence. Implementations are free to do memory accesses from a single processor in any sequence that is consistent with this partial order.

Note that accesses to different locations are ordered only with respect to barriers and TB fill. The table asymmetry for I-fetch allows writes to the I-stream to be incoherent until an IMB is executed.

Definition of Memory Access Sequence

The access sequence for a location cannot be observed directly, nor fully predicted before an actual execution, nor reproduced exactly from one execution to another. Nonetheless, some useful ordering properties must hold in all Alpha implementations.

Definition of Location Access Order

A partial ordering, called location access order, is imposed on the memory access sequence defined above.

For two accesses *u* and *v* to location *x*, *u* is said to PRECEDE *v* IN ACCESS ORDER (≪) if *u* occurs earlier than *v* in the access sequence for *x*, and at least one of them is a write:

Table 5-2 • Location Access Order

1st↓/2nd→	Pi:I(x,b)	Pi:R(x,b)	Pi:W(x,b)
Pi:I(x,a)			≪
Pi:R(x,a)			≪
Pi:W(x,a)	≪	≪	≪

Access order is thus a partial order imposed on the actual access sequence for a given location. Each location has a separate access order. There is no direct ordering relationship between accesses to different locations.

Note that reads and I-fetches are ordered only with respect to writes.

Definition of Storage

If *u* is Pi:W(x,a) , and *v* is either Pj:I(x,b) or Pj:R(x,b) , and u≪v , and no *w* Pk:W(x,c) exists such that u≪w≪v , then the value *b* returned by *v* is exactly the value *a* written by *u*.

Conversely, if *u* is Pi:W(x,a) , and *v* is either Pj:I(x,b) or Pj:R(x,b), and b=a (and *a* is distinguishable from values written by accesses other than *u*), then u≪v and for any other *w* Pk:W(x,c) either w≪u or v≪w .

The only way to communicate information between different processors is for one to write a shared location and the other to read the shared location and receive the newly written value. (In this context, the sending of an interrupt from processor Pi to processor Pj is modeled as Pi writing to a location INTij, and Pj reading from INTij.)

Relationship Between Issue Order and Access Order

If *u* is Pi:*(x,a) , and *v* is Pi:*(x,b) , one of which is a write, and u<v in the issue order for processor Pi, then u≪v in the access order for location *x*.

In other words, if two accesses to the same location are ordered on a given processor, they are ordered in the same way at the location.

Definition of Before

For two accesses *u* and *v*, *u* is said to be BEFORE v (⇐) if:

u < v or
u ≪ v or
there exists an access w such that:

(u < w and w ⇐ v) or
(u ≪ w and w ⇐ v).

In other words, "before" is the transitive closure over issue order and access order.

Definition of After

If u ⇐ v , then *v* is said to be AFTER *u*.

At most one of u ⇐ v and v ⇐ u is true.

Timeliness

Even in the absence of a barrier after the write, a write by one processor to a given location may not be delayed indefinitely in the access order for that location.

Litmus Tests

Many issues about writing and reading shared data can be cast into questions about whether a write is before or after a read. These questions can be answered by rigorously applying the ordering rules described previously to demonstrate whether the accesses in question are ordered at all.

Assume, in the litmus tests below, that initially all memory locations contain 1.

Litmus Test 1 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(x,2)
	[V2] Pj:R(x,1)

V1 reading 2 implies U1 ≪ V1, by the definition of storage
V2 reading 1 implies V2 ≪ U1, by the definition of storage
V1 < V2, by the definition of issue order

The first two orderings imply that V2 ⇐ V1 , whereas the last implies that V1 ⇐ V2 .

Both implications cannot be true. Thus, once a processor reads a new value from a location, it must never see an old value—time must not go backward. V2 must read 2.

Litmus Test 2 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)
	[V2] Pj:R(x,2)
	[V3] Pj:R(x,3)

V2 reading 2 implies V1 $\Leftarrow$ U1
V3 reading 3 implies U1 $\Leftarrow$ V1

Both implications cannot be true. Thus, once a processor reads a new value written by U1, any other writes that must precede the read must also precede U1. V3 must read 2.

Litmus Test 3 (Impossible Sequence)

Pi	Pj	Pk
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)	[W1] Pk:R(x,3)
[U2] Pi:R(x,3)		[W2] Pk:R(x,2)

U2 reading 3 implies U1 $\Leftarrow$ V1
W2 reading 2 implies V1 $\Leftarrow$ U1

Both implications cannot be true. Again, time cannot go backward. If U2 reads 3 then W2 must read 3. Alternately, if W2 reads 2, then U2 must read 2.

Litmus Test 4 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:W(y,2)	[V2] Pj:R(x,1)

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 and V2 $\Leftarrow$ U1. U1 and U2 are not ordered with respect to each other. V1 and V2 are not ordered with respect to each other. There is no conflicting implication that U1 $\Leftarrow$ V2 .

Litmus Test 5 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
	[V2] Pj:MB
[U2] Pi:W(y,2)	[V3] Pj:R(x,1)

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 $\Leftarrow$ V3 $\Leftarrow$ U1 . There is no conflicting implication that U1 $\Leftarrow$ U2 .

Litmus Test 6 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:MB	
[U3] Pi:W(y,2)	[V2] Pj:R(x,1)

There are no conflicts in this sequence. V2 ⇐ U1 ⇐ U3 ⇐ V1. There is no conflicting implication that V1 ⇐ V2.

In scenarios 4, 5, and 6, writes to two different locations x and y are observed (by another processor) to occur in the opposite order than that in which they were performed. An update to y propagates quickly to Pj, but the update to x is delayed, and Pi and Pj do not both have MBs.

Litmus Test 7 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:MB	[V2] Pj:MB
[U3] Pi:W(y,2)	[V3] Pj:R(x,1)

V1 reading 2 implies U3 ⇐ V1
V3 reading 1 implies V3 ⇐ U1
But, by transitivity, U1 ⇐ U3 ⇐ V1 ⇐ V3

Both cannot be true, so if V1 reads 2, then V3 must also read 2.

Litmus Test 8 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(y,2)
[U2] Pi:MB	[V2] Pj:MB
[U3] Pi:R(y,1)	[V3] Pj:R(x,1)

U3 reading 1 implies U3 ⇐ V1
V3 reading 1 implies V3 ⇐ U1
But, by transitivity, U1 ⇐ U3 ⇐ V1 ⇐ V3

Both cannot be true, so if U3 reads 1, then V3 must read 2, and vice versa.

Litmus Test 9 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)
[U2] Pi:R(x,2)	[V2] Pj:R(x,3)
[U3] Pi:R(x,3)	[V3] Pj:R(x,2)

V3 reading 2 implies U1 ⇐ V3
V2 ⇐ V3 and V2 reading 3 implies V2 ⇐ U1
V1 ⇐ V2 and V2 ⇐ U1 implies V1 ⇐ U1

U3 reading 3 implies V1 ⇐ U3
U2 ⇐ U3 and U2 reading 2 implies U2 ⇐ V1
U1 ⇐ U2 and U2 ⇐ V1 implies U1 ⇐ V1

Both V1 ⇐ U1 and U1 ⇐ V1 cannot be true. Time cannot go backwards. If V3 reads 2, then U3 must read 2. Alternatively, If U3 reads 3, then V3 must read 3.

Implied Barriers

In Alpha, there are no implied barriers. If an implied barrier is needed for functionally correct access to shared data, it must be written as an explicit instruction. (Software must explicitly include any needed MB or IMB instructions.)

Alpha transitions such as the following have no built-in implied memory barriers:

- Entry to PALcode
- Sending and receiving interrupts
- Returning from exceptions, interrupts, or machine checks
- Swapping context
- Invalidating the Translation Buffer (TB)

Depending on implementation choices for maintaining cache coherency, some PAL/cache implementations may have an implied IMB in the I-stream TB fill routine, but this is transparent to the non-PAL programmer.

Implications for Software

Software must explicitly include MB or IMB instructions in the following circumstances.

Single-Processor Data Stream

No barriers are ever needed. A read to physical address *x* will always return the value written by the immediately preceding write to *x* in the processor issue sequence.

Single-Processor Instruction Stream

An I-fetch from virtual or physical address *x* does not necessarily return the value written by the immediately preceding write to *x* in the issue sequence. To make the I-fetch reliably get the newly written instruction, an IMB is needed between the write and the I-fetch.

Multiple-Processor Data Stream (Including Single Processor with DMA I/O)

The only way to communicate shared data reliably is to write the shared data on one processor, then do an MB on that processor, then write a flag (equivalently, send an interrupt) signaling the other processor that the shared data is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), then do an MB, then read or update the shared data.

Leaving out the first MB removes the assurance that the shared data is written before the flag is.

Leaving out the second MB removes the assurance that the shared data is read or updated only after the flag is seen to change; in this case, an early read could see an old value, and an early update could be overwritten.

This implies that after a CPU has prepared some data buffer to be read from memory by a DMA I/O device (such as writing a buffer to disk), it must do an MB before starting the I/O, and the I/O device after receiving the start signal must logically do an MB before reading the data buffer.

This also implies that after a DMA I/O device has written some data to memory (such as paging in a page from disk), the DMA device must logically do an MB before posting a completion interrupt, and the interrupt handler software must do an MB before the data is guaranteed to be visible to the interrupted processor. Other processors must also do MBs before they are guaranteed to see the new data.

An important special case occurs when a write is done (perhaps by an I/O device) to some physical page frame, then an MB, then a previously invalid PTE is changed to be a valid mapping of the physical page frame that was just written. In this case, all processors that access using the newly valid PTE must guarantee to deliver the newly written data after the TB miss, for both I-stream and D-stream accesses.

Multiple-Processor Instruction Stream (Including Single Processor with DMA I/O)

The only way to update the I-stream reliably is to write the shared I-stream on one processor, then do an IMB (MB if the writing processor is not going to execute the new I-stream) on that processor, then write a flag (equivalently, send an interrupt) signaling the other processor that the shared I-stream is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), then do an IMB, then fetch the shared I-stream.

Leaving out the first IMB(MB) removes the assurance that the shared I-stream is written before the flag is.

Leaving out the second IMB removes the assurance that the shared I-stream is read only *after* the flag is seen to change; in this case, an early read could see an old value.

This implies that after a DMA I/O device has written some I-stream to memory (such as paging in a page from disk), the DMA device must logically do an IMB(MB) before posting a completion interrupt, and the interrupt handler software must do an IMB before the I-stream is guaranteed to be visible to the interrupted processor. Other processors must also do IMBs before they are guaranteed to see the new I-stream.

An important special case occurs when a write is done (perhaps by an I/O device) to some physical page frame, then an IMB(MB), then a previously invalid PTE is changed to be a valid mapping of the physical page frame that was just written. In this case, all processors that access using the newly valid PTE must guarantee to deliver the newly written I-stream after the TB miss.

Multiple-Processor Context Switch

If a process migrates from executing on one processor to executing on another, the context switch operating system code must include a number of barriers.

A process migrates by having its context stored into memory, then eventually having that context reloaded on another processor. In between, some shared mechanism must be used to communicate that the context saved in memory by the first processor is available to the second processor. This could be done by using an interrupt, by using a flag bit associated with the saved context, or by using a shared-memory multiprocessor data structure, as follows:

First Processor		Second Processor
:		
Save state of current process.		
MB [1]		
Pass ownership of process context data structure memory.	⇒	Pick up ownership of process context data structure memory. MB [2] Restore state of new process context data structure memory. Make I-stream coherent [3]. Make TB coherent [4]. : Execute code for new process that accesses memory that is not common to all processes.

MB [1] ensures that the writes done to save the state of the current process happen before the ownership is passed.

MB [2] ensures that the reads done to load the state of the new process happen after the ownership is picked up and hence are reliably the values written by the processor saving the old state. Leaving this MB out makes the code fail if an old value of the context remains in the second processor's cache and invalidates from the writes done on the first processor are not delivered soon enough.

The TB on the second processor must be made coherent with any write to the page tables that may have occurred on the first processor just before the save of the process state. This must be done with a series of TB invalidate instructions to remove any nonglobal page mapping for this process, or by assigning an ASN that is unused on the second processor to the process. One of these actions must occur sometime before starting execution of the code for the new process that accesses memory (instruction or data) that is not common to all processes. A common method is to assign a new ASN after gaining ownership of the new process and before loading its context, which includes its ASN.

The D-cache on the second processor must be made coherent with any write to the D-stream that may have occurred on the first processor just before the save of process state. This is ensured by MB [2] and does not require any additional instructions.

The I-cache on the second processor must be made coherent with any write to the I-stream that may have occurred on the first processor just before the save of process state. This can be done with an IMB PAL call sometime before the execution of any code that is not common to all processes, More commonly, this can be done by forcing a TB miss (via the new ASN or via TB invalidate instructions) and using the TB-fill rule (see *Multiple-Processor Data Stream (Including Single Processor with DMA I/O)* in this chapter). This latter approach does not require any additional instruction.

Combining all these considerations gives:

First Processor		Second Processor
:		
Pick up ownership of process context data structure memory.		
MB		
Assign new ASN or invalidate TBs.		
Save state of current process.		
Restore state of new process.		
MB		:
Pass ownership of process context data structure memory.	⇒	Pickup ownership of new process context data structure memory.
:		MB
:		Assign new ASN or invalidate TBs.
		Save state of current process.
		Restore state of new process.
		MB
		Pass ownership of old process context data structure memory.
		:
		Execute code for new process that accesses memory that is not common to all processes.

Note that on a single processor there is no need for the barriers.

Multiple-Processor Send/Receive Interrupt

If one processor writes some shared data, then sends an interrupt to a second processor, and that processor receives the interrupt, then accesses the shared data, the sequence from *Multiple-Processor Data Stream (Including Single Processor with DMA I/O)* in this chapter must be used:

First Processor		Second Processor
:		
Write data		
MB		
Send int.	⇒	Receive int.
		MB
		Access data
		:

Leaving out the MB at the beginning of the interrupt-receipt routine makes the code fail if an old value of the context remains in the second processor's cache and invalidates from the writes done on the first processor are not delivered soon enough.

Implications for Hardware

The coherency point for physical address x is the place in the memory subsystem at which accesses to x are ordered. It may be at a main memory board, or at a cache containing x exclusively, or at the point of winning a common bus arbitration.

The coherency point for x may move with time, as exclusive access to x migrates between main memory and various caches.

MB and IMB force all preceding writes to at least reach their respective coherency points. This does not mean that main-memory writes have been done, just that the *order* of the eventual writes is committed. For example, on the XMI with retry, this means getting the writes acknowledged as received with good parity at the inputs to memory board queues; the actual RAM write happens later.

MB and IMB also force all queued cache invalidates to be delivered to the local caches before starting any subsequent reads (that may otherwise cache hit on stale data) or writes (that may otherwise write the cache, only to have the write effectively overwritten by a late-delivered invalidate).

Implementations may allow reads of x to hit (by physical address) on pending writes in a write buffer, even before the writes to x reach the coherency point for x. If this is done, it is still true that no earlier value of x may subsequently be delivered to the processor that took the hit on the write buffer value.

Virtual data caches are allowed to deliver data before doing address translation, but only if there cannot be a pending write under a synonym virtual address. Lack of a write-buffer match on untranslated address bits is sufficient to guarantee this.

Virtual data caches must invalidate or otherwise become coherent with the new value whenever a PALcode routine is executed that affects the validity, fault behavior, protection behavior, or virtual-to-physical mapping specified for one or more pages. Becoming coherent can be delayed until the next subsequent MB instruction or TB fill (using the new mapping), if the implementation of the PALcode routine always forces a subsequent TB fill.

Arithmetic Traps

Alpha implementations are allowed to execute multiple instructions concurrently and to forward results from one instruction to another. Thus, when an arithmetic trap is detected, the PC may have advanced an arbitrarily large number of instructions past the instruction T (calculating result R) whose execution triggered the trap.

When the trap is detected, any or all of these subsequent instructions may run to completion before the trap is actually taken. Instruction T and the set of instructions subsequent to T that complete before the trap is taken are collectively called the trap shadow of T. The PC pushed on the stack when the trap is taken is the PC of the first instruction past the trap shadow.

The instructions in the trap shadow of T may use the undefined result R of T, they may generate additional traps, and they may completely change the PC (branches, JSR).

Thus, by the time a trap is taken, the PC pushed on the stack may bear no useful relationship to the PC of the trigger instruction T, and the state visible to the programmer may have been updated using the undefined result R. If an instruction in the trap shadow of T uses R to calculate a subsequent register value, that register value is undefined, even though there may be no trap associated with the subsequent calculation. Similarly:

- If an instruction in the trap shadow of T stores R or any subsequent undefined result, the stored value is undefined.
- If an instruction in the trap shadow of T uses R or any subsequent undefined result as the basis of a conditional or calculated branch, the branch target is undefined.
- If an instruction in the trap shadow of T uses R or any subsequent undefined result as the basis of an address calculation, the memory address actually accessed is undefined.

Software that is intended to bound how far the PC may advance before taking a trap, or how far an undefined result may propagate, must insert TRAPB instructions at appropriate points.

Software that is intended to continue from a trap by supplying a well-defined result R within an arithmetic trap handler, can do so reliably by following the rules for software completion code sequences given in *Floating-Point Trapping Modes* in Chapter 4.

Chapter 6 • Common PALcode Architecture

• PALcode

In a family of machines, both users and operating system implementors require functions to be implemented consistently. When functions conform to a common interface, the code that uses those functions can be used on several different implementations without modification.

These functions range from the binary encoding of the instruction and data to the exception mechanisms and synchronization primitives. Some of these functions can be implemented cost effectively in hardware, but others are impractical to implement directly in hardware. These functions include low-level hardware support functions such as Translation Buffer miss fill routines, interrupt acknowledge, and vector dispatch. They also include support for privileged and atomic operations that require long instruction sequences.

In the VAX, these functions are generally provided by microcode. This is not seen as a problem because the VAX architecture lends itself to a microcoded implementation.

One of the goals of Alpha is that microcode will not be necessary for practical implementation. However, it is still desirable to provide an architected interface to these functions that will be consistent across the entire family of machines. The Privileged Architecture Library (PALcode) provides a mechanism to implement these functions without resorting to a microcoded machine.

• PALcode Environment

The PALcode environment differs from the normal environment in the following ways:

- Complete control of the machine state.
- Interrupts are disabled.
- Implementation-specific hardware functions are enabled, as described below.
- I-stream memory management traps are prevented (by disabling I-stream mapping, mapping PALcode with a permanent TB entry, or by other mechanisms).

Complete control of the machine state allows all functions of the machine to be controlled. Disabling interrupts allows the system to provide multi-instruction sequences as atomic operations. Enabling implementation-specific hardware functions allows access to low-level system hardware. Preventing I-stream memory management traps allows PALcode to implement memory management functions such as Translation Buffer fill.

• Special Functions Required for PALcode

PALcode uses the Alpha instruction set for most of its operations. A small number of additional functions are needed to implement the PALcode. There are five opcodes reserved to implement PALcode functions: PALRES0, PALRES1, PALRES2, PALRES3 and PALRES4. These instructions produce an Illegal Instruction Trap if executed outside the PALcode environment.

- PALcode needs a mechanism to save the current state of the machine and dispatch into PALcode.
- PALcode needs a set of instructions to access hardware control registers.
- PALcode needs a hardware mechanism to transition the machine from the PALcode environment to the non-PALcode environment. This mechanism loads the PC, enables interrupts, enables mapping, and disables PALcode privileges.

An Alpha implementation may also choose to provide additional functions to simplify or improve performance of some PALcode functions. The following are some examples:

- An Alpha implementation may include a read/write virtual function that allows PALcode to perform mapped memory accesses using the mapping hardware rather than providing the virtual-to-physical translation in PALcode routines. PALcode may provide a special function to do physical reads and writes and have the Alpha loads and stores continue to operate on virtual address in the PALcode environment.
- An Alpha implementation may include hardware assists for various functions—for example, saving the virtual address of a reference on a memory management error rather than having to generate it by simulating the effective address calculation in PALcode.
- An Alpha implementation may include private registers so it can function without having to save and restore the native general registers.

• PALcode Effects on System Code

PALcode will have one effect on system code. Because PALcode may be resident in main memory and maintain privileged data structures in main memory, the operating system code that allocates physical memory cannot use all of physical memory.

The amount of memory PALcode requires is small, so the loss to the system is negligible.

• PALcode Replacement

Alpha systems are required to support the replacement of Digital-supplied PALcode with an operating system-specific version. The following functions must be implemented in PALcode, *not* directly in hardware, to facilitate replacement with different versions.

1. Translation Buffer fill. Different operating systems will want to replace the Translation Buffer (TB) fill routines. The replacement routines will use different data structures. Therefore, no portion of the TB fill flow that would change with a change in page tables may be placed in hardware, unless it is placed in a manner that can be overridden by PALcode.
2. Process structure. Different operating systems might want to replace the process context switch routines. The replacement routines will use different data structures. Therefore, no portion of the context switching flows that would change with a change in process structure may be placed in hardware.

PALcode must be written in a modular manner that facilitates easy replacement of major subsections. The subsections that need to be simple to replace are:

- Translation Buffer fill
- Process structure and context switch
- Interrupt and exception frame format and routine dispatch
- Privileged PALcode instructions

Required PALcode Instructions

The PALcode instructions listed in Table 6-1 and described in the following sections must be supported by all Alpha implementations:

Table 6-1 • Required PALcode Instructions

Mnemonic	Type	Operation
HALT	Privileged	Halt processor
IMB	Unprivileged	I-stream memory barrier

Halt

Format:

```
CALL_PAL HALT                    !PALcode format
```

Operation:

```
IF  PS<CM> NE 0  THEN
    {privileged instruction exception}
CASE {halt_action} OF
    halt:                {halt}
    restart/halt:        {restart/halt}
    restart/boot/halt:   {restart/boot/halt}
    boot/halt:           {boot/halt}
ENDCASE
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

CALL_PAL HALT Halt Processor

Description:

The HALT instruction stops normal instruction processing, and depending on the HALT action setting, the processor may either enter console mode or the restart sequence.

Instruction Memory Barrier

Format:

```
CALL_PAL IMB                    !PALcode format
```

Operation:

```
{Make instruction stream coherent with Data stream}
```

Exceptions:

None

Instruction mnemonics:

CALL_PAL IMB I-stream Memory Barrier

Description:

An IMB instruction must be executed after software or I/O devices write into the instruction stream or modify the instruction stream virtual address mapping, and before the new value is fetched as an instruction. An implementation may contain an instruction cache that does not track either processor or I/O writes into the instruction stream. The instruction cache and memory are made coherent by an IMB instruction.

If the instruction stream is modified and an IMB is not executed before fetching an instruction from the modified location, it is UNPREDICTABLE whether the old or new value is fetched.

The cache coherency and sharing rules are described in Chapter 5.

Chapter 7 • Console Subsystem Overview

On an Alpha system, underlying control of the system platform hardware is provided by a *console*. The console:

1. Initializes, tests, and prepares the system platform hardware for Alpha system software.
2. Bootstraps (loads into memory and starts the execution of) system software.
3. Controls and monitors the state and state transitions of each processor in a multiprocessor system.
4. Provides services to system software that simplify system software control of and access to platform hardware.
5. Provides a means for a *console operator* to monitor and control the system.

The console interacts with system platform hardware to accomplish the first three tasks. The actual mechanisms of these interactions are specific to the platform hardware; however, the net effects are common to all systems.

The console interacts with system software once control of the system platform hardware has been transferred to that software.

The console interacts with the console operator through a virtual display device or *console terminal*. The console operator may be a human being or a management application.

Chapter 8 • Alpha VMS

The following sections specify the Privileged Architecture Library (PALcode) instructions, that are required to support an Alpha VMS system.

• Unprivileged VMS PALcode Instructions

The unprivileged PALcode instructions provide support for system operations to all modes of operation (Kernel, Executive, Supervisor, and User).

Table 8-1 describes the unprivileged VMS PALcode instructions.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary

Mnemonic	Operation and Description
BPT	Breakpoint
	The BPT instruction is provided for program debugging. It switches the processor to Kernel mode and pushes R2..R7, the updated PC, and PS on the Kernel stack. It then dispatches to the address in the Breakpoint vector, stored in a control block.
BUGCHK	Bugcheck
	The BUGCHK instruction is provided for error reporting. It switches the processor to Kernel mode and pushes R2..R7, the updated PC, and PS on the Kernel stack. It then dispatches to the address in the Bugcheck vector, stored in a control block.
CHME	Change mode to Executive
	The CHME instruction allows a process to change its mode in a controlled manner.
	A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. Registers R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction.
CHMK	Change mode to Kernel
	The CHMK instruction allows a process to change its mode to Kernel in a controlled manner.
	A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the Kernel stack. The saved PC addresses the instruction following the CHMK instruction.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
CHMS	Change mode to Supervisor
	The CHMS instruction allows a process to change its mode in a controlled manner.
	A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.
CHMU	Change mode to User
	The CHMU instruction allows a process to call a routine via the change mode mechanism.
	R2..R7, PS, and PC are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction.
GENTRAP	Generate trap
	The GENTRAP instruction is provided for reporting runtime software conditions. It switches the processor to Kernel mode and pushes registers R2..R7, the updated PC, and the PS on the Kernel stack. It then dispatches to the address of the GENTRAP vector, stored in a control block.
IMB	I-Stream memory barrier
	The IMB instruction ensures that the contents of an instruction cache are coherent after the instruction stream has been modified by software or I/O devices.
	If the instruction stream is modified and an IMB is not executed before fetching an instruction from the modified location, it is UNPREDICTABLE whether the old or new value is fetched.
INSQHIL	Insert into longword queue at header, interlocked
	The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
INSQHILR	Insert into longword queue at header, interlocked resident The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
INSQHIQ	Insert into quadword queue at header, interlocked The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQHIQR	Insert into quadword queue at header, interlocked resident The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
INSQTIL	Insert into longword queue at tail, interlocked The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQTILR	Insert into longword queue at tail, interlocked resident The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
INSQTIQ	Insert into quadword queue at tail, interlocked
	The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
INSQTIQR	Insert into quadword queue at tail, interlocked resident
	The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.
	This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
INSQUEL	Insert into longword queue
	The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEL, or following the entry specified by the contents of the longword addressed by R16 for INSQUEL/D. The insertion is a noninterruptible operation.
INSQUEQ	Insert into quadword queue
	The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEQ, or following the entry specified by the contents of the quadword addressed by R16 for INSQUEQ/D. The insertion is a noninterruptible operation.
PROBE	Probe read/write access
	PROBE checks the read (PROBER) or write (PROBEW) accessibility of the first and last byte specified by the base address and the signed offset; the bytes in between are not checked. System software must check all pages between the two bytes if they are to be accessed.
	PROBE is only intended to check a single datum for accessibility.
RD_PS	Read processor status
	RD_PS writes the Processor Status (PS) to register R0.
READ_UNQ	Read unique context
	READ_UNQ reads the hardware process (thread) unique context value, if previously written by WRITE_UNQ, and places that value in R0.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
REI	Return from exception or interrupt
	The PS, PC, and saved R2..R7 are popped from the current stack and held in temporary registers. The new PS is checked for validity and consistency. If it is valid and consistent, the current stack pointer is then saved and a new stack pointer is selected. Registers R2 through R7 are restored by using the saved values held in the temporary registers. A check is made to determine if an AST or interrupt is pending.
	If the enabling conditions are present for an interrupt or AST at the completion of this instruction, the interrupt or AST occurs before the next instruction.
REMQHIL	Remove from longword queue at header, interlocked
	The self-relative queue entry following the header, pointed to by R16, is removed from the queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQHILR	Remove from longword queue at header, interlocked resident
	The queue entry following the header, pointed to by R16, is removed from the self-relative queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
	This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
REMQHIQ	Remove from quadword queue at header, interlocked
	The self-relative queue entry following the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQHIQR	Remove from quadword queue at header, interlocked resident
	The queue entry following the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
	This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
REMQTIL	Remove from longword queue at tail, interlocked
	The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQTILR	Remove from longword queue at tail, interlocked resident
	The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
	This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.
REMQTIQ	Remove from quadword queue at tail, interlocked
	The self-relative queue entry preceding the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
REMQTIQR	Remove from quadword queue at tail, interlocked resident
	The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.
	This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.
REMQUEL	Remove from longword queue
	The queue entry addressed by R16 for REMQUEL or the entry addressed by the longword addressed by R16 for REMQUEL/D is removed from the longword absolute queue, and the address of the removed entry is returned in R1. The removal is a noninterruptible operation.

Table 8-1 • Unprivileged VMS PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
REMQUEQ	Remove from quadword queue
	The queue entry addressed by R16 for REMQUEQ or the entry addressed by the quadword addressed by R16 for REMQUEL/D is removed from the quadword absolute queue, and the address of the removed entry removed is returned in R1. The removal is a noninterruptible operation.
RSCC	Read system cycle counter
	Register R0 is written with the value of the system cycle counter. This counter is an unsigned 64-bit integer that increments at the same rate as the process cycle counter.
	The system cycle counter is suitable for timing a general range of intervals to within 10% error and may be used for detailed performance characterization.
SWASTEN	Swap AST enable
	SWASTEN swaps the AST enable bit for the current mode. The new state for the enable bit is supplied in register R16<0> and previous state of the enable bit is returned, zero-extended, in R0.
	A check is made to determine if an AST is pending. If the enabling conditions are present for an AST at the completion of this instruction, the AST occurs before the next instruction.
WRITE_UNQ	Write unique context
	WRITE_UNQ writes the hardware process (thread) unique context value passed in R16 to internal storage or to the hardware privileged context block.
WR_PS_SW	Write processor status software field
	WR_PS_SW writes the Processor Status software field (PS<SW>) with the low-order three bits of R16<2:0>.

• Privileged VMS Palcode Instructions

The privileged PALcode instructions can be called in Kernel mode only.

Table 8-2 describes the privileged VMS PALcode instructions.

Table 8-2 • Privileged VMS PALcode Instructions Summary

Mnemonic	Operation and Description
CFLUSH	Cache flush
	At least the entire physical page specified by a page frame number in R16 is flushed from any data caches associated with the current processor. After doing a CFLUSH, the first subsequent load on the same processor to an arbitrary address in the target page is fetched from physical memory.
DRAINA	Drain aborts
	DRAINA stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.
HALT	Halt processor
	The HALT instruction stops normal instruction processing.
LDQP	Load quadword physical
	The quadword-aligned memory operand, whose physical address is in R16, is fetched and written to R0.
	If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.
MFPR	Move from processor register
	The internal processor register specified by the PALcode function field is written to R0.
MTPR	Move to processor register
	The source operands in integer registers R16 (and R17, reserved for future use) are written to the internal processor register specified by the PALcode function field. The effect of loading a processor register is guaranteed to be active on the next instruction.
STQP	Store quadword physical
	The quadword contents of R17 are written to the memory location whose physical address is in R16.
	If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.
SWPCTX	Swap privileged context
	The SWPCTX instruction returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.

Chapter 9 • Alpha OSF/1

The following sections specifiy the Privileged Architecture Library (PALcode) instructions that are required to support an Alpha OSF/1 system.

• Unprivileged OSF/1 PALcode Instructions

Table 9-1 describes the unprivileged OSF/1 PALcode instructions.

Table 9-1 • Unprivileged OSF/1 PALcode Instruction Summary

Mnemonic	Operation and Description
bpt	Break Point Trap
	The bpt instruction switches mode to Kernel, builds a stack frame on the Kernel stack, and dispatches to the breakpoint code.
bugchk	Bugcheck
	The bugchk instruction switches mode to Kernel, builds a stack frame on the Kernel stack, and dispatches to the breakpoint code.
callsys	System Call
	The callsys instruction switches mode to Kernel, builds a callsys stack frame, and dispatches to the system call code.
gentrap	Generate Trap
	The gentrap instruction switches mode to Kernel, builds a stack frame on the Kernel stack, and dispatches to the gentrap code.
imb	I-Stream Memory Barrier
	The imb instruction makes the I-cache coherent with main memory.
rdunique	Read Unique
	The rdunique instruction returns the process unique value.
wrunique	Write Unique
	The wrunique instruction sets the process unique register.

• Privileged OSF/1 PALcode Instructions

The privileged PALcode instructions can be called only from Kernel mode. They provide an interface to control the privileged state of the machine.

Table 9-2 describes the privileged OSF/1 PALcode instructions.

Table 9-2 • Privileged OSF/1 PALcode Instruction Summary

Mnemonic	Operation and Description
halt	Halt Processor
	The halt instruction stops normal instruction processing. Depending on the halt action setting, the processor can either enter console mode or the restart sequence.
rdps	Read Processor Status
	The rdps instruction returns the current PS.
rdusp	Read User Stack Pointer
	The rdusp instruction reads the User stack pointer while in Kernel mode and returns it.
rdval	Read System Value
	The rdval instruction reads a 64-bit per-processor value and returns it.
retsys	Return from System Call
	The retsys instruction pops the return address, the User stack pointer, and the User global pointer from the Kernel stack. It then saves the Kernel stack pointer, sets mode to User, enables interrupts, and jumps to the address popped off the stack.
rti	Return from Trap, Fault or Interrupt
	The rti instruction pops certain registers from the Kernel stack. If the new mode is User, the Kernel stack is saved and the User stack restored.
swpctx	Swap Privileged Context
	The swpctx instruction saves the current process data in the current process control block (PCB). Then swpctx switches to the PCB and loads the new process context.
swpipl	Swap IPL
	The swpipl instruction returns the current value IPL and sets the IPL.
tbi	TB invalidate
	The tbi instruction removes entries from the instruction and data translation buffers when the mapping entries change.

Table 9-2 • Privileged OSF/1 PALcode Instruction Summary (Continued)

Mnemonic	Operation and Description
whami	Who_Am_I
	The whami instruction returns the processor number for the current processor. The processor number is in the range 0 to the number of processors minus one (0..numproc–1) that can be configured in the system.
wrent	Write System Entry Address
	The wrent instruction sets the virtual address of the system entry points.
wrfen	Write Floating-Point Enable
	The wrfen instruction writes a bit to the floating-point enable register.
wrkgp	Write Kernel Global Pointer
	The wrkgp instruction writes the Kernel global pointer internal register.
wrusp	Write User Stack Pointer
	The wrusp instruction writes a value to the User stack pointer while in Kernel mode.
wrval	Write System Value
	The wrval instruction writes a 64-bit per-processor value.
wrvptptr	Write Virtual Page Table Pointer
	The wrvptptr instruction writes a pointer to the virtual page table pointer (vptptr).

Appendix A • Software Considerations

• Hardware-Software Compact

The Alpha architecture, like all RISC architectures, depends on careful attention to data alignment and instruction scheduling to achieve high performance.

Since there will be various implementations of the Alpha architecture, it is not obvious how compilers can generate high-performance code for all implementations. This chapter gives some scheduling guidelines that, if followed by all compilers and respected by all implementations, will result in good performance. As such, this section represents a good-faith compact between hardware designers and software writers. It represents a set of common goals, not a set of architectural requirements. Thus, an Appendix, not a Chapter.

Many of the performance optimizations discussed below are advantageous only for frequently executed code. For rarely executed code, they may produce a bigger program that is not any faster. Some of the branching optimizations also depend on good prediction of which path from a conditional branch is more frequently executed. These optimizations are best done by using an execution profile, either an estimate generated by compiler heuristics, or a real profile of a previous run, such as that gathered by PC-sampling in PCA.

Each computer architecture has a "natural word size." For the PDP-11, it is 16 bits; for VAX, 32 bits; and for Alpha, 64 bits. Other architectures also have a natural word size that varies between 16 and 64 bits. Except for very low-end implementations, ALU data paths, cache access paths, chip pin buses, and main memory data paths are all usually the natural word size.

As an architecture becomes commercially successful, high-end implementations inevitably move to double-width data paths that can transfer an *aligned* (at an even natural word address) pair of natural words in one cycle. For Alpha, this means eventual 128-bit wide data paths. It is hard to get much speed advantage from paired transfers unless the code being executed has instructions and data appropriately aligned on aligned octaword boundaries. Since this is hard to retrofit to old code, the following sections sometimes encourage "over-aligning" to octaword boundaries in anticipation of high-speed Alpha implementations.

In some cases, there are performance advantages in aligning instructions or data to cache-block boundaries, or putting data whose use is correlated into the same cache block, or trying to avoid cache conflicts by not having data whose use is correlated placed at addresses that are equal modulo the cache size. Since the Alpha architecture will have many implementations, an exact cache design cannot be outlined here. Nonetheless, some expected bounds can be stated.

1. Small (first-level) cache sizes will likely be in the range 2 KB to 64 KB
2. Small cache block sizes will likely be 16, 32, 64, or 128 bytes
3. Large (second- or third-level) cache sizes will likely be in the range 128 KB to 8 MB
4. Large cache block sizes will likely be 32, 64, 128, or 256 bytes
5. TB sizes will likely be in the range 16 to 1024 entries

Thus, if two data items need to go in different cache blocks, it is desirable to make them at least 128 bytes apart (modulo 2 KB). Doing that creates a high probability of allowing both items to be in a small cache simultaneously, for all Alpha implementations.

In each case below, the performance implication is given by an order-of-magnitude number: 1, 3, 10, 30, or 100. A factor of 10 means that the performance difference being discussed will likely range from 3 to 30 across all Alpha implementations.

Instruction-Stream Considerations

The following sections describe considerations for the instruction stream.

Instruction Alignment

Code PSECTs should be octaword-aligned. Targets of frequently taken branches should be at least quadword-aligned, and octaword-aligned for very frequent loops. Compilers could use execution profiles to identify frequently taken branches.

Most Alpha implementations will fetch aligned quadwords of instruction stream (two instructions), and many will waste an instruction-issue cycle on a branch to an odd longword. High-end implementations may eventually fetch aligned octawords, and waste up to 3 issue cycles on a branch to an odd longword. Some implementations may only be able to fetch wide chunks of instructions every other CPU cycle. Fetching four instructions from an aligned octaword can get at most one cache miss, while fetching them from an odd longword address can get 2 or even 3 cache misses.

Quadword I-fetch implementors should give first priority to executing aligned quadwords quickly. Octaword-fetch implementors should give first priority to executing aligned octawords quickly, and second priority to executing aligned quadwords quickly. Dual-issue implementations should give first priority to issuing both halves of an aligned quadword in one cycle, and second priority to buffering and issuing other combinations.

Multiple Instruction Issue—Factor of 3

Some Alpha implementations will issue multiple instructions in a single cycle. To improve the odds of multiple-issue, compilers should choose pairs of instructions to put in aligned quadwords. Pick one from column A and one from column B (but only a total of one load/store/branch per pair).

Column A	Column B
Integer Operate	Floating Operate
Floating Load/Store	Integer Load/Store
Floating Branch	Integer Branch
	BR/BSR/JSR

Implementors of multiple-issue machines should give first priority to dual-issuing at least the above pairs, and second priority to multiple-issue of other combinations.

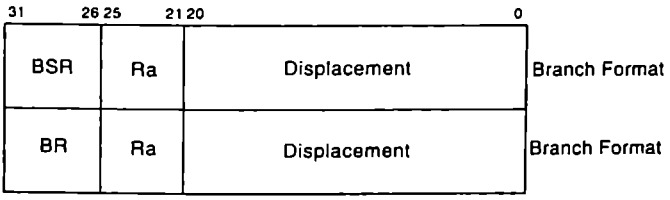
In general, the above rules will give a good hardware-software match, but compilers may want to implement model-specific switches to generate code tuned more exactly to a specific implementation.

Branch Prediction and Minimizing Branch-Taken—Factor of 3

In many Alpha implementations, an unexpected change in I-stream address will result in about 10 lost instruction times. "Unexpected" may mean any branch-taken or may mean a mispredicted branch. In many implementations, even a correctly predicted branch to a quadword target address will be slower than straight-line code.

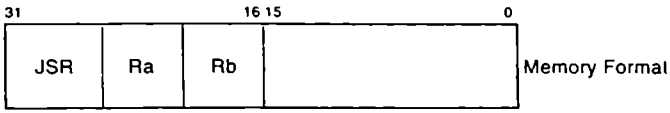
Compilers should follow these rules to minimize unexpected branches:

1. Implementations will predict all forward conditional branches as not-taken, and all backward conditional branches as taken. Based on execution profiles, compilers should physically rearrange code so that it has matching behavior.
2. Make basic blocks as big as possible. A good goal is 20 instructions on average between branch-taken. This means unrolling loops so that they contain at least 20 instructions, and putting subroutines of less than 20 instructions directly in line. It also means using execution profiles to rearrange code so that the frequent case of a conditional branch falls through. For very high-performance loops, it will be profitable to move instructions across conditional branches to fill otherwise wasted instruction issue slots, even if the instructions moved will not always do useful work. Note that the Conditional Move instructions can sometimes be used to avoid breaking up basic blocks.
3. In an if-then-else construct whose execution profile is skewed even slightly away from 50%–50% (51–49 is enough), put the infrequent case completely out of line, so that the frequent case encounters *zero* branch-takens, and the infrequent case encounters *two* branch-takens. If the infrequent case is rare (5%), put it far enough away that it never comes into the I-cache. If the infrequent case is extremely rare (error message code), put it on a page of rarely executed code and expect that page *never* to be paged in.
4. There are two functionally identical branch-format opcodes, BSR and BR.



Compilers should use the first one for subroutine calls, and the second for GOTOs. Some implementations may push a stack of predicted return addresses for BSR and not push the stack for BR. Failure to compile the correct opcode will result in mispredicted return addresses, and hence make subroutine returns slow.

5. The memory-format JSR instruction has 16 unused bits. These should be used by the compilers to communicate a hint about expected branch-target behavior (see Chapter 4):



   If the JSR is used for a computed GOTO or a CASE statement, compile bits <15:14> as 00, and bits <13:0> such that (updated PC+Instr<13:0>*4) <15:0> equals (likely_target_addr) <15:0>. In other words, pick the low 14 bits so that a normal PC+displacement*4 calculation will match the low 16 bits of the most likely target longword address. (Implementations will likely prefetch from the matching cache block.)

   If the JSR is used for a computed subroutine call, compile bits <15:14> as 01, and bits <13:0> as above. Some implementations will prefetch the call target using the prediction and also push updated PC on a return-prediction stack.

   If the JSR is used as a subroutine return, compile bits <15:14> as 10. Some implementations will pop an address off a return-prediction stack.

   If the JSR is used as a coroutine linkage, compile bits <15:14> as 11. Some implementations will pop an address off a return-prediction stack and also push updated PC on the return-prediction stack.

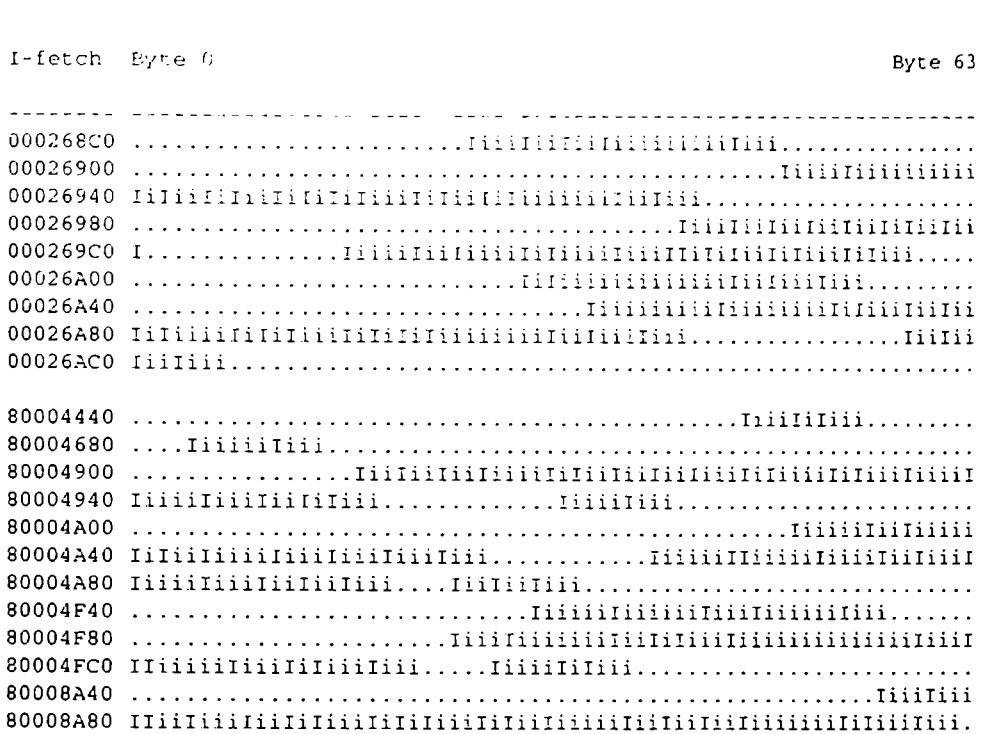
Implementors should give first priority to executing straight-line code with no branch-takens as quickly as possible, second priority to predicting conditional branches based on the sign of the displacement field (backward taken, forward not-taken), and third priority to predicting subroutine return addresses by running a small prediction stack. (VAX traces show a stack of 2 to 4 entries correctly predicts most branches.)

Improving I-Stream Density—Factor of 3

Compilers should try to use profiles to make sure almost 100 percent of the bytes brought into an I-cache are actually executed. This means aligning branch targets and putting rarely executed code out of line. Doing so would consistently make an I-cache appear about two times larger, compared to current VAX practice.

The example below shows the bytes actually brought into a VAX cache (from part of an address trace of a DLINPAC). The dots represent bytes brought into the cache but never executed. They occupy about half of the cache.

Each line shows the use of an aligned 64-byte I-cache block. A portion of DLINPAC and a portion of VMS 4.x are shown. Uppercase I is the first byte of an instruction, and lowercase i marks subsequent bytes. Period (.) shows a byte brought into the cache but never executed.



Instruction Scheduling—Factor of 3

The performance of Alpha programs will be sensitive to how carefully the code is scheduled to minimize instruction-issue delays.

"Result latency" is defined as the number of CPU cycles that must elapse between an instruction that writes a result register and one that uses that register, if execution-time stalls are to be avoided. Thus, a latency of zero means that the instruction writes a result register and the instruction that uses that register can be multiple-issued in the *same* cycle. A latency of 2 means that if the writing instruction is issued at cycle N, the reading instruction can issue no earlier than cycle N+2. Latency is implementation-specific.

Most Alpha instructions have a non-zero result latency. Compilers should schedule code so that a result is not used too soon, at least in frequently executed code (inner loops, as identified by execution profiles). In general, this will require loop unrolling and short procedure inlining.

"Too soon" is currently ill-defined, since no implementations have been designed yet. For starters, assume that implementations can dual-issue instructions. Assume that Load and JSR instructions have a latency of 3, shifts and byte manipulation a latency of 2, integer multiply a latency of 10, and other integer operates a latency of 1. Assume floating multiply has a latency of 5, floating divide a latency of 10, and other floating operates a latency of 4. Scheduling to these latencies will give at least reasonable performance on currently anticipated implementations.

Compilers should try to schedule code to match the above latency rules and also to match the multiple-issue rules. If doing both is impractical for a particular sequence of code, the latency rules are more important (since they apply even in single-issue implementations).

Implementors should give first priority to minimizing the latency of back-to-back integer operations, of address calculations immediately followed by load/store, of load immediately followed by branch, and of compare immediately followed by branch. Second priority should be given to minimizing latencies in general.

• Data-Stream Considerations

The following sections describe considerations for the data stream.

Data Alignment—Factor of 10

Data PSECTs should be at least octaword-aligned, so that aggregates (arrays, some records, subroutine stack frames) can be allocated on aligned octaword boundaries to take advantage of any implementations with aligned octaword data paths, and to decrease the number of cache fills in almost all implementations.

Aggregates (arrays, records, common blocks, and so forth) should be allocated on at least aligned octaword boundaries whenever language rules allow this. In some implementations, a series of writes that completely fill a cache block may be a factor of 10 faster than a series of writes that partially fill a cache block, when that cache block would give a read miss. This is true of writeback caches that read a partially filled cache block from memory, but optimize away the read for completely filled blocks.

For such implementations, long strings of sequential writes will be faster if they start on a cache-block boundary (a multiple of 128 bytes will do well for most, if not all, Alpha implementations). This applies to array results that sweep through large portions of memory, and also to register-save areas for context switching, graphics frame buffer accesses, and other places where exactly 8, 16, 32, or more quadwords are stored sequentially. Allocating the targets at multiples of 8, 16, 32, or more quadwords, respectively, and doing the writes in order of increasing address will maximize the write speed.

Items within aggregates that are forced to be unaligned (records, common blocks) should generate compile-time warning messages and inline byte extract/insert code. Users must be educated that the warning message means that they are taking a factor of 30 performance hit.

Compilers should consider supplying a switch that allows the compiler to pad aggregates to avoid unaligned data.

Compiled code for parameters should assume that the parameters are aligned. Unaligned actuals will therefore cause runtime alignment traps and very slow fixups. The fixup routine, if invoked, should generate warning messages to the user, preferably giving the first few statement numbers that are doing unaligned parameter access, and at the end of a run the total number of alignment traps (and perhaps an estimate of the performance improvement if the data were aligned). Again, users must be educated that the trap routine warning message means they are taking a factor of 30 performance hit.

Frequently used scalars should reside in registers. Each scalar datum allocated in memory should normally be allocated an aligned quadword to itself, even if the datum is only a byte wide. This allows aligned quadword loads and stores and avoids partial-quadword writes (which may be half as fast as full-quadword writes, due to such factors as read-modify-write a quadword to do quadword ECC calculation).

Implementors should give first priority to fast reads of aligned octawords and second priority to fast writes of full cache blocks. Partial-quadword writes need not have a fast repetition rate.

Shared Data in Multiple Processors—Factor of 3

Software locks are aligned quadwords and should be allocated to large cache blocks that either contain no other data, or read-mostly data whose usage is correlated with the lock.

Whenever there is high contention for a lock, one processor will have the lock and be using the guarded data, while other processors will be in a read-only spin loop on the lock bit. Under these circumstances, *any* write to the cache block containing the lock will likely cause excess bus traffic and cache fills, thus having a performance impact on all processors that are involved, and the buses between them. In some decomposed FORTRAN programs, refills of the cache blocks containing one or two frequently used locks can account for a third of all the bus bandwidth the program consumes.

Whenever there is almost no contention for a lock, one processor will have the lock and be using the guarded data. Under these circumstances, it might be desirable to keep the guarded data in the *same* cache block as the lock.

For the high sharing case, compilers should assume that *almost all* accesses to shared data result in cache misses all the way back to main memory, for each distinct cache block used. Such accesses will likely be a factor of 30 slower than cache hits. It is helpful to pack correlated shared data into a small number of cache blocks. It is helpful also to segregate blocks written by one processor from blocks read by others.

Therefore, accesses to shared data, including locks, should be minimized. For example, a 4-processor decomposition of some manipulation of a 1000-row array should avoid accessing lock variables every row, but instead might access a lock variable every 250 rows.

Array manipulation should be partitioned across processors so that cache blocks do not thrash between processors. Having each of 4 processors work on every fourth array element severely impairs performance on any implementation with a cache block of 4 elements or larger. The processors all contend for copies of the *same* cache blocks and use only 1/4 of the data in each block. Writes in one processor severely impair cache performance on all processors.

A better decomposition is to give each processor the largest possible contiguous chunk of data to work on (N/4 consecutive rows for 4 processors and row-major array storage; N/4 columns for column-major storage). With the possible exception of 3 cache blocks at the partition boundaries, this decomposition will result in each processor caching data that is touched by *no* other processor.

Operating-system scheduling algorithms should attempt to minimize process migration from one processor to another. Any time migration occurs, there are likely to be a large number of cache misses on the new processor.

Similarly, operating-system scheduling algorithms should attempt to enforce some affinity between a given device's interrupts and the processor on which the interrupt-handler runs. I/O control data structures and locks for different devices should be disjoint. Doing both of these allows higher cache hit rates on the corresponding I/O control data structures.

Implementors should give first priority to an efficient (low-bandwidth) way of transferring isolated lock values and other isolated, shared write data between processors.

Implementors should assume that the amount of shared data will continue to increase, so over time the need for efficient sharing implementations will also increase.

Avoiding Cache/TB Conflicts—Factor of 1

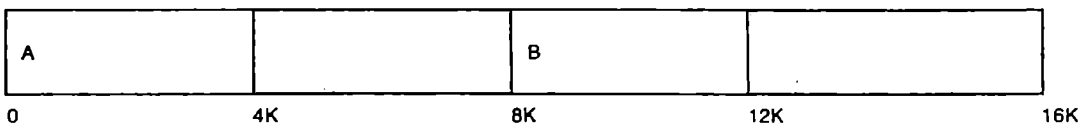
Occasionally, programs that run with a direct-mapped cache or TB will thrash, taking excessive cache or TB misses. With some work, thrashing can be minimized at compile time.

In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in *exactly the same* aligned 64-byte block, or differ in bits VA<10:6>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 64 bytes of padding *between* the arrays. This rule will avoid thrashing in small direct-mapped data caches with block sizes up to 64 bytes and total sizes of 2K bytes or more.

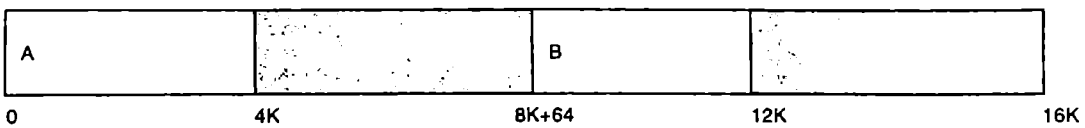
Example:

```
   REAL*4 A(1000),B(1000)
   DO 60 i=1,1000
60 A(i) = f(B(i))
```

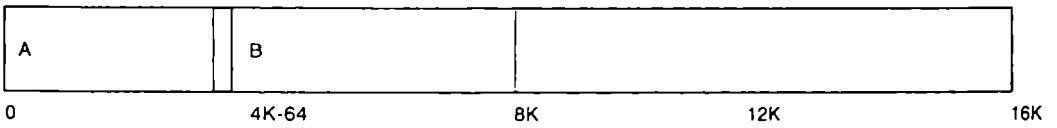
BAD allocation (A and B thrash in 8 KB direct-mapped cache):



BETTER allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously):



BEST allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously, *and* both arrays fit entirely in 8 KB or bigger cache):



In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in *exactly the same 8 KB* page, or differ in bits VA<17:13>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 8K bytes of padding *between* the arrays. This rule will avoid thrashing in direct-mapped TBs and in some large direct-mapped data caches, with total sizes of 32 pages (256 KB) or more.

Usually, this padding will mean *zero* extra bytes in the executable image, just a skip in virtual address space to the next-higher page boundary.

For large caches, the rule above should be applied to the I-stream, in addition to all the D-stream references. Some implementations will have combined I-stream/D-stream large caches.

Both of the rules above can be satisfied simultaneously, thus often eliminating thrashing in all anticipated direct-mapped cache/TB implementations.

Sequential Read/Write—Factor of 1

All other things being equal, sequences of consecutive reads or writes should use ascending (rather than descending) memory addresses. Where possible, the memory address for a block of 2**Kbytes should be on a 2**K boundary, since this minimizes the number of different cache blocks used and minimizes the number of partially written cache blocks.

To avoid overrunning memory bandwidth, sequences of more than eight quadword Loads or Stores should be broken up with intervening instructions (if there is any useful work to be done).

For consecutive reads, implementors should give first priority to prefetching ascending cache blocks, and second priority to absorbing up to eight consecutive quadword Loads (aligned on a 64-byte boundary) without stalling.

For consecutive writes, implementors should give first priority to avoiding read overhead for fully written aligned cache blocks, and second priority to absorbing up to eight consecutive quadword Stores (aligned on a 64-byte boundary) without stalling.

Prefetching—Factor of 3

To use FETCH and FETCH_M effectively, software should follow this programming model:

1. Assume that at most two FETCH instructions can be outstanding at once, and that there are two prefetch address registers, PREa and PREb, to hold prefetching state. FETCH instructions alternate between loading PREa and PREb. Each FETCH instruction overwrites any previous prefetching state, thus terminating any previous prefetch that is still in progress in the register that is loaded. The order of fetching within a block and the order between PREa and PREb are UNPREDICTABLE.

   Implementation Note

   Implementations are encouraged to alternate at convenient intervals between PREa and PREb.

2. Assume, for maximum efficiency, that there should be about 64 unrelated memory access instructions (load or store) between a FETCH and the first actual data access to the prefetched data.
3. Assume, for instruction-scheduling purposes in a multilevel cache hierarchy, that FETCH does not prefetch data to the innermost cache level, but rather one level out. Schedule loads to bury the last level of misses.
4. Assume that FETCH is worthwhile if, on average, at least half the data in a block will be accessed. Assume that FETCH_M is worthwhile if, on average, at least half the data in a block will be modified.
5. Treat FETCH as a vector load. If a piece of code could usefully prefetch 4 operands, launch the first two prefetches, do about 128 memory references worth of work, then launch the next two prefetches, do about 128 more memory references worth of work, then start using the 4 sets of prefetched data.
6. Treat FETCH as having the same effect on a cache as a series of 64 quadword loads. If the loads would displace useful data, so will FETCH. If two sets of loads from specific addresses will thrash in a direct-mapped cache, so will two FETCH instructions using the same pair of addresses.

   Implementation Note

   Hardware implementations are expected to provide either no support for FETCHx or support that closely matches this model.

• Code Sequences

The following section describes code sequences.

Aligned Byte/Word Memory Accesses

The instruction sequences given in Chapter 4 for byte and word accesses are worst-case code. In the common case of accessing a byte or aligned word field at a known offset from a pointer that is expected to be at least longword aligned, the common-case code is much shorter.

"Expected" means that the code should run fast for a longword-aligned pointer and trap for unaligned. The trap handler may at its option fix up the unaligned reference.

For access at a known offset D from a longword-aligned pointer Rx, let D.lw be D rounded down to a multiple of 4 ((D div 4)*4), and let D.mod be D mod 4 .

In the common case, the intended sequence for loading and zero-extending an aligned word is:

```
LDL     R1,D.lw(Rx)      ! Traps if unaligned
EXTWL   R1,#D.mod,R1     ! Picks up word at byte 0 or byte 2
```

In the common case, the intended sequence for loading and sign-extending an aligned word is:

```
LDL     R1,D.lw(Rx)         ! Traps if unaligned
SLL     R1,#48-8*D.mod,R1   ! Aligns word at high end of R1
SRA     R1,#48,R1           ! SEXT to low end of R1
```

Note

The shifts often can be combined with shifts that might surround subsequent arithmetic operations (for example, to produce word overflow from the high end of a register).

In the common case, the intended sequence for loading and zero-extending a byte is:

```
LDL     R1,D.lw(Rx)      !
EXTBL   R1,#D.mod,R1     !
```

In the common case, the intended sequence for loading and sign-extending a byte is:

```
LDL     R1,D.lw(Rx)         !
SLL     R1,#56-8*D.mod,R1   !
SRA     R1,#56,R1           !
```

In the common case, the intended sequence for storing an aligned word R5 is:

```
LDL     R1,D.lw(Rx)      !
INSWL   R5,#D.mod,R3     !
MSKWL   R1,#D.mod,R1     !
BIS     R3,R1,R1         !
STL     R1,D.lw(Rx)      !
```

In the common case, the intended sequence for storing a byte R5 is:

```
LDL     R1,D.lw(Rx)      !
INSBL   R5,#D.mod,R3     !
MSKBL   R1,#D.mod,R1     !
BIS     R3,R1,R1         !
STL     R1,D.lw(Rx)      !
```

Division

In all implementations, floating-point division is likely to have a substantially longer result latency than floating-point multiply; in addition, in many implementations multiplies will be pipelined and divides will not.

Thus, any division by a constant power of two should be compiled as a multiply by the exact reciprocal, if it is representable without overflow or underflow. If language rules or surrounding context allow, other divisions by constants can be closely approximated via multiplication by the reciprocal.

Integer division does not exist as a hardware opcode. Division by a constant can always be done via UMULH of another appropriate constant, followed by a right shift. General quadword division by true variables can be done via a subroutine. The subroutine could test for small divisors (less than about 1000 in absolute value) and for those, do a table lookup on the exact constant and shift count for an UMULH/shift sequence. For the remaining cases, a table lookup on about a 1000-entry table and a multiply can give a linear approximation to 1/divisor that is accurate to 16 bits. Using this approximation, a multiply and a back-multiply and a subtract can generate one 16-bit quotient "digit" plus a 48-bit new partial dividend. Three more such steps can generate the full quotient. Having prior knowledge of the possible sizes of the divisor and dividend, normalizing away leading bytes of zeros, and performing an early-out test can reduce the average number of multiplies to about 5 (compared to a best case of 1 and a worst case of 9).

Stylized Code Forms

Using the same stylized code form for a common operation makes compiler output a little more readable and makes it more likely that an implementation will speed up the stylized form.

NOP

The standard NOP forms are:

```
NOP             ==      BIS     R31,R31,R31
FNOP            ==      CPYS    F31,F31,F31
```

These generate no exceptions. In most implementations, they should encounter no operand issue delays, no destination issue delay, and no functional unit issue delay. Implementations are free to optimize these into no action and zero execution cycles.

Clear a Register

The standard clear register forms are:

```
CLR             ==      BIS     R31,R31,Rx
FCLR            ==      CPYS    F31,F31,Fx
```

These generate no exceptions. In most implementations, they should encounter no operand issue delays, and no functional unit issue delay.

Load Literal

The standard load integer literal (ZEXT 8-bit) form is:

```
MOV #lit8,Ry    ==    BIS R31, lit8, Ry
```

The Alpha literal construct in Operate instructions creates a canonical longword constant for values 0..255.

A longword constant stored in an Alpha 64-bit register is in canonical form when bits <63:32>=bit <31>.

A canonical 32-bit literal can usually be generated with one or two instructions, but sometimes three instructions are needed. Use the following procedure to determine the offset fields of the instructions:

```
val  = <sign-extended, 32-bit value>

low  = val<15:0>
tmp1 = val - SEXT(low)! Account for LDA instruction

high = tmp1<31:16>
tmp2 = tmp1 - SHIFT_LEFT( SEXT(high,16) )

if tmp2 NE 0 then
     ! original val was in range 7FFF8000₁₆..7FFFFFFF₁₆
            extra = 4000₁₆
            tmp1 = tmp1 - 40000000₁₆
            high = tmp1<31:16>
else
     extra = 0
endif
```

The general sequence is:

```
LDA  Rdst, low(R31)
LDAH Rdst, extra(Rdst)   ! Omit if extra=0
LDAH Rdst, high(Rdst)    ! Omit if high=0
```

Register-to-Register Move

The standard register move forms are:

```
MOV RX,RY   ==   BIS   RX,RX,RY
FMOV FX,FY  ==   CPYS  FX,FX,FY
```

These generate no exceptions. In most implementations, these should encounter no functional unit issue delay.

Negate

The standard register negate forms are:

```
NEGz Rx,Ry      ==      SUBz    R31,Rx,Ry     ! z = L or Q
NEGz Fx,Fy      ==      SUBz    F31,Fx,Fy     ! z = F G S or T

FNEGz Fx,Fy     ==      CPYSN   Fx,Fx,Fy      ! z = F G S or T
```

The integer subtract generates no Integer Overflow trap if Rx contains the largest negative number (SUBz/V would trap). The floating subtract generates a floating-point exception for a non-finite value in Fx. The CPYSN form generates no exceptions.

NOT

The standard integer register NOT form is:

```
NOT Rx,Ry       ==      ORNOT   R31,Rx,Ry
```

This generates no exceptions. In most implementations, this should encounter no functional unit issue delay.

Booleans

The standard alternative to BIS is:

```
OR Rx,Ry,Rz     ==      BIS     Rx,Ry,Rz
```

The standard alternative to BIC is:

```
ANDNOT Rx,Ry,Rz ==      BIC     Rx,Ry,Rz
```

The standard alternative to EQV is:

```
XORNOT Rx,Ry,Rz ==      EQV     Rx,Ry,Rz
```

Trap Barrier

The TRAPB instruction guarantees that following instructions do not issue until all possible preceding traps have been signaled. This does not mean that all preceding instructions have necessarily run to completion (for example, a Load instruction may have passed all the fault checks but not yet delivered data from a cache miss).

Pseudo-Operations (Stylized Code Forms)

This section summarizes the pseudo-operations for the Alpha architecture that may be used by various software components in an Alpha system. Most of these forms are discussed in preceding sections.

In the context of this section, pseudo-operations all represent a single underlying machine instruction. Each pseudo-operation represents a particular instruction with either replicated fields (such as FMOV), or hard-coded zero fields. Since the pattern is distinct, these pseudo-operations can be decoded by instruction decode mechanisms.

In Table A-1, the pseudo-operation codes can be viewed as macros with parameters. The formal form is listed in the left column, and the expansion in the code stream listed in the right column.

Some instruction mnemonics have synonyms. These are different from pseudo-operations in that each synonym represents the same underlying instruction with no special encoding of operand fields. As a result, synonyms cannot be distinquished from each other. They are not listed in the table that follows. Examples of synonyms are: BIC/ANDNOT, BIS/OR, and EQV/XORNOT.

Table A-1 • Decodable Pseudo-Operations (Stylized Code Forms)

Pseudo-Operation in Listing		Actual Instruction Encoding	
No-exception generic floating absolute value:			
FABS	Fx, Fy	CPYS	F31, Fx, Fy
Branch to target (21-bit signed displacement):			
BR	target	BR	R31, target
Clear integer register:			
CLR	Rx	BIS	R31, R31, Rx
Clear a floating-point register:			
FCLR	Fx	CPYS	F31, F31, Fx
Floating-point move:			
FMOV	Fx, Fy	CPYS	Fx, Fx, Fy
No-exception generic floating negation:			
FNEG	Fx, Fy	CPYSN	Fx, Fx, Fy
Floating-point no-op:			
FNOP		CPYS	F31, F31, F31
Move Rx/8-bit zero-extended literal to Ry:			
MOV	{Rx/Lit8}, Ry	BIS	R31, {Rx/Lit8}, Ry

Table A-1 • Decodable Pseudo-Operations (Stylized Code Forms) (Continued)

Pseudo-Operation in Listing		Actual Instruction Encoding	
Move 16-bit sign-extended literal to Rx:			
MOV	Lit, Rx	LDA	Rx, lit(R31)
Move to FPCR:			
MT_FPCR	Fx	MT_FPCR	Fx, Fx, Fx
Move from FPCR:			
MF_FPCR	Fx	MF_FPCR	Fx, Fx, Fx
Negate F_floating:			
NEGF	Fx, Fy	SUBF	F31, Fx, Fy
Negate F_floating, semi-precise:			
NEGF/S	Fx, Fy	SUBF/S	F31, Fx, Fy
Negate G_floating:			
NEGG	Fx, Fy	SUBG	F31, Fx, Fy
Negate G_floating, semi-precise:			
NEGG/S	Fx, Fy	SUBG/S	F31, Fx, Fy
Negate longword:			
NEGL	{Rx/Lit8}, Ry	SUBL	R31, {Rx/Lit}, Ry
Negate longword with overflow detection:			
NEGL/V	{Rx/Lit8}, Ry	SUBL/V	R31, {Rx/Lit}, Ry
Negate quadword:			
NEGQ	{Rx/Lit8}, Ry	SUBQ	R31, {Rx/Lit}, Ry
Negate quadword with overflow detection:			
NEGQ/V	{Rx/Lit8}, Ry	SUBQ/V	R31, {Rx/Lit}, Ry
Negate S_floating:			
NEGS	Fx, Fy	SUBS	F31, Fx, Fy
Negate S_floating, software with underflow detection:			
NEGS/SU	Fx, Fy	SUBS/SU	F31, Fx, Fy
Negate S_floating, software with underflow and inexact result detection:			
NEGS/SUI	Fx, Fy	SUBS/SUI	F31, Fx, Fy
Negate T_floating:			
NEGT	Fx, Fy	SUBT	F31, Fx, Fy

Table A-1 • Decodable Pseudo-Operations (Stylized Code Forms) (Continued)

Pseudo-Operation in Listing		Actual Instruction Encoding	
Negate T_floating, software with underflow detection:			
NEGT/SU	Fx, Fy	SUBT/SU	F31, Fx, Fy
Negate T_floating, software with underflow and inexact result detection:			
NEGT/SUI		SUBT/SUI	F31, Fx, Fy
Integer no-op:			
NOP		BIS	R31, R31, R31
Logical NOT of Rx/8-bit zero-extended literal storing results in Ry:			
NOT	{Rx/Lit8}, Ry	ORNOT	R31, {Rx/Lit}, Ry
Longword sign-extension of Rx storing results in Ry:			
SEXTL	{Rx/Lit8}, Ry	ADDL	R31, {Rx/Lit}, Ry

• Timing Considerations: Atomic Sequences

A sufficiently long instruction sequence between LDx_L and STx_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The following rules describe sequences that will eventually complete in all Alpha implementations:

1. At most 40 operate or conditional-branch (not taken) instructions executed in the sequence between LDx_L and STx_C.
2. At most two I-stream TB-miss faults. Sequential instruction execution guarantees this.
3. No other exceptions triggered during the last execution of the sequence.

Implementation Note

On all expected implementations, this allows for about 50 μsec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1 msec timer interrupt rate.

Appendix B • IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (754–1985) is provided in the Alpha floating-point instructions. This appendix describes how to construct a complete IEEE implementation.

The order of presentation parallels the order of the IEEE specification.

• Alpha Choices for IEEE Options

Alpha supports IEEE single and double formats. Optional extended double is not supported.

Alpha hardware supports normal and chopped IEEE rounding modes. IEEE plus infinity and minus infinity rounding modes can be implemented in hardware or software.

Alpha hardware does not support optional IEEE software trap enable/disable modes; see the following discussion about software support.

Alpha hardware supports add, subtract, multiply, divide, convert between floating formats, convert between floating and integer formats, and compare. Software routines support square root, remainder, round to integer in floating-point format, and convert binary to/from decimal.

In the Alpha architecture, copying without change of format is not considered an operation. (LDx, CPYSx, and STx do not check for non-finite numbers; an operation would.) Compilers may generate ADDx F31,Fx,Fy to get the opposite effect.

Optional operations for differing formats are not provided.

The Alpha choice is that the accuracy provided will meet or exceed IEEE standard requirements. It is implementation-dependent whether the software binary/decimal conversions beyond 9 or 17 digits treat any excess digits as zeros.

Overflow and underflow, NaNs, and infinities encountered during software binary to decimal conversion return strings that specify the conditions. Such strings can be truncated to their shortest unambiguous length.

Alpha hardware supports comparisons of same-format numbers. Software supports comparisons of different-format numbers.

In the Alpha architecture, results are true-false in response to a predicate.

Alpha hardware supports the required six predicates and the optional unordered predicate. The other 19 optional predicates can be constructed from sequences of two comparisons and two branches.

Alpha hardware supports infinity arithmetic only by trapping when an infinity operand is encountered and when an infinity is to be created from finite operands by overflow or division by zero. A software trap handler (interposed between the hardware and the IEEE user) provides correct infinity arithmetic.

Alpha hardware supports NaNs only by trapping when a NaN operand is encountered and when a NaN is to be created. A software trap handler (interposed between the hardware and the IEEE user) provides correct Signaling and Quiet NaN behavior.

In the Alpha architecture, Quiet NaNs do not afford retrospective diagnostic information.

In the Alpha architecture, copying a Signaling NaN without a change of format does not signal an invalid exception (LDx, CPYSx, and STx do not check for non-finite numbers). Compilers may generate ADDx F31,Fx,Fy to get the opposite effect.

Alpha hardware fully supports negative zero operands, and follows the IEEE rules for creating negative zero results.

Alpha hardware does not supply IEEE exception trap behavior; the hardware traps are a superset of the IEEE-required conditions. A software trap handler (interposed between the hardware and the IEEE user) provides correct IEEE exception behavior.

In the Alpha architecture, tininess is detected by hardware after rounding, and loss of accuracy is detected by software as an inexact result.

In the Alpha architecture, user trap handlers will be supported by compilers and a software trap handler (interposed between the hardware and the IEEE user), as described in the next section.

• Alpha Hardware Support of Software Exception Handlers

In Alpha instructions, hardware trap behavior is determined only at compile time; short of recompiling, there are no dynamic facilities for changing hardware trap behavior.

There is an essential disparity between the Alpha design goal of fast execution and the IEEE design goal of exact trap behavior. The Alpha hardware architecture provides means for users to choose various degrees of IEEE compliance, at appropriate performance cost.

Instructions compiled without the /Software modifier cannot produce IEEE-compliant trap behavior, nor can they provide IEEE-compliant non-finite arithmetic. Trapping and stopping on non-finite operands or results (rather than the IEEE default of continuing with NaNs propagated) is an Alpha value-added behavior that some users prefer.

Instructions compiled without the /Underflow hardware trap enable modifier cannot produce IEEE-compliant underflow trap behavior, nor can they provide IEEE-compliant denormal results. They are fast and provide true zero (not minus zero) results whenever underflow occurs. This is an Alpha value-added behavior that some users prefer.

Instructions compiled without the /Inexact hardware trap enable modifier cannot produce IEEE-compliant inexact trap behavior. Trapping on Inexact will be painfully slow; few users appear to prefer this, but they can get it if they really want it.

IEEE floating-point instructions compiled with the /Software modifier produce hardware traps and unpredictable values; a software trap handler may then produce all IEEE-required behavior.

IEEE floating-point instructions compiled with the /Underflow enable modifier produce hardware traps and true zero values for underflow; a software trap handler may then produce all IEEE-required behavior.

IEEE floating-point instructions compiled with the /Inexact enable modifier produce hardware traps that allow a software trap handler to produce all IEEE-required behavior.

Thus, to get full IEEE compliance of all the required features of the standard, users must compile with all three options enabled.

To get the optional full IEEE user trap handler behavior, a software trap handler must be provided that implements the five exception flags, dynamic user trap handler disabling, handler saving and restoring, default behavior for disabled user trap handlers, and linkages that allow a user handler to return a substitute result.

Also, users must insert a TRAPB in every basic block with a floating operation that can potentially trap, so that a software handler has an opportunity to scale the true result by 2**192 or 2**1536, as appropriate for enabled user trap handlers; and to supply the default +/– infinity, +/–MAX, +/–MIN, denormal, or zero as appropriate for disabled user trap handlers.

• Mapping to IEEE Standard

There are five IEEE exceptions, each of which can be "IEEE software trap-enabled" or disabled (the default condition). Implementing the IEEE software trap-enabled mode is optional in the IEEE standard.

Our assumption, therefore, is that the only access to IEEE-specified software trap-enabled results will be generated in assembly language code. The following design allows this, but *only* if such assembly language code has TRAPB instructions after each floating-point instruction, and generates the IEEE-specified scaled result in a trap handler by emulating the instruction that was trapped by hardware overflow/underflow detection, using the original operands.

There is a set of detailed IEEE-specified result values, both for operations that are specified to raise IEEE traps and those that do not. This behavior is created on Alpha by four layers of hardware, PALcode, the operating-system trap handler, and the user IEEE trap handler, as shown in Figure B-1.

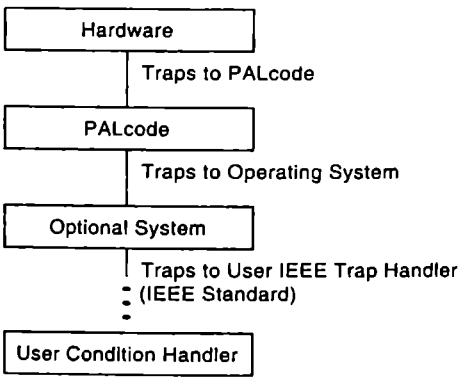


Figure B-1 • IEEE Trap Handling Behavior

The IEEE-specified trap behavior occurs *only* with respect to the user IEEE trap handler (the last layer in Figure B-1); any trap-and-fixup behavior in the first three layers is outside the scope of the IEEE standard.

The IEEE number system is divided into finite and non-finite numbers:

- The finites are normal numbers:

  –MAX..–MIN, –0, 0, +MIN..+MAX

- The non-finites are:

  Denormals, +/– Infinity, Signaling NaN, Quiet NaN

Alpha hardware must treat minus zero operands and results as special cases, as required by the IEEE standard.

Table B-1 specifies, for the IEEE /Software modes, which layer does each piece of trap handling. See Chapter 4 for more detail on the hardware instruction descriptions.

Table B-1 • IEEE Floating-Point Trap Handling

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
FBEQ FBNE FBLT FBLE FBGT FBGE	Bits Only—No Exceptions			
LDS LDT	Bits Only—No Exceptions			
STS STT	Bits Only—No Exceptions			
CPYS CPYSN	Bits Only—No Exceptions			
FCMOVx	Bits Only—No Exceptions			
ADDx SUBx INPUT Exceptions				
Denormal operand	Trap	Trap	Supply sum	–
+/–Inf operand	Trap	Trap	Supply sum	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
+Inf + –Inf	Trap	Trap	Supply QNaN	[Invalid Op]

Table B-1 • IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
ADDx SUBx OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–[1]
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled in the instruction	–	–	–	–
Inexact and enabled in the instruction	Trap	Trap	–	[Inexact]
MULx INPUT Exceptions				
Denormal operand	Trap	Trap	Supply prod.	–
+/–Inf operand	Trap	Trap	Supply prod.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
0 * Inf	Trap	Trap	Supply QNaN	[Invalid Op]
MULx OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and Trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	Trap	Trap	–	[Inexact]

[1] An implementation could choose instead to trap to PALcode and have the PALcode supply a zero result on all underflows.

Table B-1 • IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
DIVx INPUT Exceptions				
Denormal operand	Trap	Trap	Supply quot.	–
+/–Inf operand	Trap	Trap	Supply quot.	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
0/0 or Inf/Inf	Trap	Trap	Supply QNaN	[Invalid Op]
A/0	Trap	Trap	Supply +/–Inf	[Div. Zero]
DIVx OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	Trap	Trap	–	[Inexact]
CMPTEQ CMPTUN INPUT Exceptions				
Denormal operand	Trap	Trap	Supply (=)	–
QNaN operand	Trap	Trap	Supply False for EQ, True for UN	–
SNaN operand	Trap	Trap	Supply False/ True	[Invalid Op]

Table B-1 • IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CMPTLT CMPTLE INPUT Exceptions				
Denormal operand	Trap	Trap	Supply (=)	–
QNaN operand	Trap	Trap	Supply False	[Invalid Op]
SNaN operand	Trap	Trap	Supply False	[Invalid Op]
CVTFi INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/–Inf operand	Trap	Trap	Supply Cvt	[Invalid Op]
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
CVTFi OUTPUT Exceptions				
Inexact and disabled	–	–	–	–
Inexact and enabled	Trap	Trap	–	[Inexact]
Integer overflow	Supply Trunc. result and trap if enabled	Trap	–	[Invalid Op] [2]
CVTif OUTPUT Exceptions				
Inexact and disabled	–	–	–	–
Inexact and enabled	Trap	Trap	–	[Inexact]

[2] An implementation could choose instead to trap to PALcode on extreme values and have the PALcode supply a truncated result on all overflows.

Table B-1 • IEEE Floating-Point Trap Handling (Continued)

Alpha Instructions	Hardware	PAL	OS Trap Handler	User Software Handler
CVTff INPUT Exceptions				
Denormal operand	Trap	Trap	Supply Cvt	–
+/–Inf operand	Trap	Trap	Supply Cvt	–
QNaN operand	Trap	Trap	Supply QNaN	–
SNaN operand	Trap	Trap	Supply QNaN	[Invalid Op]
CVTff OUTPUT Exceptions				
Exponent overflow	Trap	Trap	Supply +/–Inf +/–MAX	[Overflow] Scale by 2**Alpha
Exponent underflow and disabled	Supply +0	–	–	–
Exponent underflow and enabled	Supply +0 and trap	Trap	Supply +/–MIN denorm +/–0	[Underflow] Scale by 2**Alpha
Inexact and disabled	–	–	–	–
Inexact and enabled	Trap	Trap	–	[Inexact]

Other IEEE operations (software subroutines or sequences of instructions), are listed here for completeness:

Remainder
SQRT
Round float to integer-valued float
Convert binary to/from decimal
Compare, other combinations than the four above

Table B-2 shows the IEEE standard charts.

Table B-2 • IEEE Standard Charts

Exception	IEEE Software TRAP Disabled (IEEE Default)	IEEE Software TRAP Enabled (Optional)
Invalid Operation		
(1) Input signaling NaN	Quiet NaN	
(2) Mag. subtract Inf.	Quiet NaN	
(3) 0 * Inf.	Quiet NaN	
(4) 0/0 or Inf/Inf	Quiet NaN	
(5) x REM 0 or Inf REM y	Quiet NaN	
(6) SQRT(negative non-zero)	Quiet NaN	
(7) Cvt to int(ovfl, Inf, NaN)	Quiet NaN	
(8) Compare unordered	Quiet NaN	
Division by Zero		
x/0, x finite <>0	+/–Inf	
Overflow		
Round nearest	+/–Inf.	Res/2**192 or 1536
Round to zero	+/–MAX	Res/2**192 or 1536
Round to –Inf	+MAX/–Inf	Res/2**192 or 1536
Round to +Inf	+Inf/–MAX	Res/2**192 or 1536
Underflow	0/denorm/+ –MIN	Res*2**192 or 1536
Inexact	Rounded/ovfl	Res

IEEE software trap handler requirements are as follows:

Result is unpredictable unless supplied by trap handler.
Determine which exceptions occurred.
Determine the kind of operation.
Determine the destination format.
Overflow/underflow/inexact: the correctly rounded result, including parts that do not fit in the format.
Invalid and divzero: the operand values.

Appendix C • Instruction Encodings

The encodings for the Alpha instruction set are given in the following sections. There is one section for each instruction format, followed by a summary of all the instruction opcodes in a single table.

• Memory Format Instructions

Table C-1 lists the hexadecimal values of the 6-bit opcode field for the Memory format instructions.

Table C-1 • Memory Format Instruction Opcodes

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
LDL	28	STL	2C	LDF	20	STF	24
LDQ	29	STQ	2D	LDG	21	STG	25
LDL_L	2A	STL_C	2E	LDS	22	STS	26
LDQ_L	2B	STQ_C	2F	LDT	23	STT	27
LDQ_U	0B	STQ_U	0F				
LDA	08	LDAH	09				

Table C-2 lists the hexadecimal values of the 6-bit opcode field and the 16-bit displacement field for the Memory format instructions that use the displacement field as a function code. The notation used is oo.ffff , where *oo* is the 6-bit opcode and the *ffff* is the 16-bit displacement field.

Table C-2 • Memory Format Instructions with a Function Code

Mnemonic		Mnemonic		Mnemonic	
FETCH	18.8000	FETCH_M	18.A000	MB	18.4000
RC	18.E000	RPCC	18.C000	RS	18.F000
TRAPB	18.0000				

Programming Note

The code points 18.4400, 18.4800, and 18.4C00 must operate as Memory Barrier instructions (MB 18.4000). Software will currently only use the 18.4000 code point for MB. This allows a weaker memory barrier to be added.

Table C-3 lists the hexadecimal values of the high-order two bits of the displacement field for the Memory format branch instructions. The notation used is oo.h, where *oo* is the 6-bit opcode and the *h* is the high-order two bits of the displacement field.

Table C-3 • Memory Format Branch Instruction Opcodes

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
JMP	1A.0	JSR	1A.1	JSR_COROUTINE	1A.3	RET	1A.2

• Branch Format Instructions

Table C-4 lists the hexadecimal values of the 6-bit opcode field for the Branch format instructions.

Table C-4 • Branch Format instruction Opcodes

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
BR	30	FBEQ	31	FBLT	32	FBLE	33
BSR	34	FBNE	35	FBGE	36	FBGT	37
BLBC	38	BEQ	39	BLT	3A	BLE	3B
BLBS	3C	BNE	3D	BGE	3E	BGT	3F

• Operate Format Instructions

Table C-5 lists the hexadecimal values of the 6-bit opcode field and the 7-bit function code field for the Operate format instructions The notation used is oo.ff, where *oo* is the 6-bit opcode and the *ff* is the 7-bit function code field

Table C-5 • Operate Format Instruction Opcodes and Function Codes

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
ADDL	10.00	SUBL	10.09	CMPEQ	10.2D		
ADDL/V	10.40	SUBL/V	10.49	CMPLT	10.4D		
ADDQ	10.20	SUBQ	10.29	CMPLE	10.6D		
ADDQ/V	10.60	SUBQ/V	10.69	CMPULT	10.1D		
CMPULE	10.3D						
CMPBGE	10.0F						
S4ADDL	10.02	S4SUBL	10.0B	S8ADDL	10.12	S8SUBL	10.1B
S4ADDQ	10.22	S4SUBQ	10.2B	S8ADDQ	10.32	S8SUBQ	10.3B
AND	11.00	BIS	11.20	XOR	11.40		
BIC	11.08	ORNOT	11.28	EQV	11.48		
CMOVEQ	11.24	CMOVLT	11.44	CMOVLE	11.64		
CMOVNE	11.26	CMOVGE	11.46	CMOVGT	11.66		
CMOVLBS	11.14	CMOVLBC	11.16				

Table C-5 • Operate Format Instruction Opcodes and Function Codes (Continued)

Mnemonic		Mnemonic		Mnemonic		Mnemonic
SLL	12.39	SRA	12.3C	SRL	12.34	
EXTBL	12.06	INSBL	12.0B	MSKBL	12.02	
EXTWL	12.16	INSWL	12.1B	MSKWL	12.12	
EXTLL	12.26	INSLL	12.2B	MSKLL	12.22	
EXTQL	12.36	INSQL	12.3B	MSKQL	12.32	
EXTWH	12.5A	INSWH	12.57	MSKWH	12.52	
EXTLH	12.6A	INSLH	12.67	MSKLH	12.62	
EXTQH	12.7A	INSQH	12.77	MSKQH	12.72	
				ZAP	12.30	
				ZAPNOT	12.31	
MULL	13.00	MULL/V	13.40	MULQ	13.20	
MULQ/V	13.60	UMULH	13.30			

• Floating-Point Operate Format

Table C-6 lists the hexadecimal values of the 11-bit function code field for the Floating-point Operate format instructions that are data type independent. The 6-bit opcode for these instructions is 17_{16}.

Table C-6 • Function Codes for Floating Data Type Independent Operations

Mnemonic		Mnemonic		Mnemonic	
CPYS	020	CPYSN	021	CPYSE	022
MF_FPCR	025	MT_FPCR	024	CVTQL/SV	530
CVTLQ	010	CVTQL	030	CVTQL/V	130
FCMOVEQ	02A	FCMOVLT	02C	FCMOVLE	02E
FCMOVNE	02B	FCMOVGE	02D	FCMOVGT	02F

IEEE Floating-Point Instructions

Table C-7 lists the hexadecimal value of the 11-bit function code field for the IEEE floating-point instructions, with and without qualifiers. The opcode for these instructions is 16_{16}.

Table C-7 • IEEE Floating-Point Instruction Function Codes

	None	/C	/M	/D	/U	/UC	/UM	/UD
ADDS	080	000	040	0C0	180	100	140	1C0
ADDT	0A0	020	060	0E0	1A0	120	160	1E0
CMPTEQ	0A5							
CMPTLT	0A6							
CMPTLE	0A7							
CMPTUN	0A4							

Table C-7 • IEEE Floating-Point Instruction Function Codes (Continued)

	None	/C	/M	/D	/U	/UC	/UM	/UD
CVTQS	0BC	03C	07C	0FC				
CVTQT	0BE	03E	07E	0FE				
CVTTS	0AC	02C	06C	0EC	1AC	12C	16C	1EC
DIVS	083	003	043	0C3	183	103	143	1C3
DIVT	0A3	023	063	0E3	1A3	123	163	1E3
MULS	082	002	042	0C2	182	102	142	1C2
MULT	0A2	022	062	0E2	1A2	122	162	1E2
SUBS	081	001	041	0C1	181	101	141	1C1
SUBT	0A1	021	061	0E1	1A1	121	161	1E1
	/SU	/SUC	/SUM	/SUD	/SUI	/SUIC	/SUIM	/SUID
ADDS	580	500	540	5C0	780	700	740	7C0
ADDT	5A0	520	560	5E0	7A0	720	760	7E0
CMPTEQ	5A5							
CMPTLT	5A6							
CMPTLE	5A7							
CMPTUN	5A4							
CVTQS					7BC	73C	77C	7FC
CVTQT					7BE	73E	77E	7FE
CVTTS	5AC	52C	56C	5EC	7AC	72C	76C	7EC
DIVS	583	503	543	5C3	783	703	743	7C3
DIVT	5A3	523	563	5E3	7A3	723	763	7E3
MULS	582	502	542	5C2	782	702	742	7C2
MULT	5A2	522	562	5E2	7A2	722	762	7E2
SUBS	581	501	541	5C1	781	701	741	7C1
SUBT	5A1	521	561	5E1	7A1	721	761	7E1
	None	/C	/V	/VC	/SV	/SVC	/SVI	/SVIC
CVTTQ	0AF	02F	1AF	12F	5AF	52F	7AF	72F
	D	/VD	/SVD	/SVID	/M	/VM	/SVM	/SVIM
CVTTQ	0EF	1EF	5EF	7EF	06F	16F	56F	76F

Programming Note

Since underflow cannot occur for CMPTxx, there is no difference in function or performance between CMPTxx/S and CMPTxx/SU. It is intended that software generate CMPTxx/SU in place of CMPTxx/S.

VAX Floating-Point Instructions

Table C-8 lists the hexadecimal value of the 11-bit function code field for the VAX floating-point instructions. The opcode for these instructions is 15_{16}.

Table C-8 • VAX Floating-Point Instruction Function Codes

	None	/C	/U	/UC	/S	/SC	/SU	/SUC
ADDF	080	000	180	100	480	400	580	500
CVTDG	09E	01E	19E	11E	49E	41E	59E	51E
ADDG	0A0	020	1A0	120	4A0	420	5A0	520
CMPGEQ	0A5				4A5			
CMPGLT	0A6				4A6			
CMPGLE	0A7				4A7			
CVTGF	0AC	02C	1AC	12C	4AC	42C	5AC	52C
CVTGD	0AD	02D	1AD	12D	4AD	42D	5AD	52D
CVTQF	0BC	03C						
CVTQG	0BE	03E						
DIVF	083	003	183	103	483	403	583	503
DIVG	0A3	023	1A3	123	4A3	423	5A3	523
MULF	082	002	182	102	482	402	582	502
MULG	0A2	022	1A2	122	4A2	422	5A2	522
SUBF	081	001	181	101	481	401	581	501
SUBG	0A1	021	1A1	121	4A1	421	5A1	521
	None	**/C**	**/V**	**/VC**	**/S**	**/SC**	**/SV**	**/SVC**
CVTGQ	0AF	02F	1AF	12F	4AF	42F	5AF	52F

• Required PALcode Function Codes

The opcodes listed in Table C-9 are required for all Alpha implementations. The notation used is oo.ffff, where *oo* is the hexadecimal 6-bit opcode and *ffff* is the hexadecimal 26-bit function code.

Table C-9 • Required PALcode Function Codes

Mnemonic	Type	Function Code
HALT	Privileged	00.0000
IMB	Unprivileged	00.0086

• Opcodes Reserved to PALcode

The opcodes listed in Table C-10 are reserved for use in implementing PALcode.

Table C-10 • Opcodes Reserved for PALcode

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
PAL19	19	PAL1B	1B	PAL1D	1D	PAL1E	1E
PAL1F	1F						

• Opcodes Reserved to Digital

The opcodes listed in Table C-11 are reserved to Digital.

Table C-11 • Opcodes Reserved for Digital

Mnemonic		Mnemonic		Mnemonic		Mnemonic	
OPC01	01	OPC02	02	OPC03	03	OPC04	04
OPC05	05	OPC06	06	OPC07	07	OPC0A	0A
OPC0C	0C	OPC0D	0D	OPC0E	0E	OPC14	14
OPC1C	1C						

• Opcode Summary

Table C-12 lists all Alpha opcodes from 00 (CALL_PALL) through 3F (BGT). In the table, the column headings appearing over the instructions have a granularity of 8_{16}. The rows beneath the leftmost column supply the individual hex number to resolve that granularity.

If an instruction column has a 0 in the right (low) hex digit, replace that 0 with the number to the left of the backslash in the leftmost column on the instruction's row. If an instruction column has an 8 in the right (low) hexadecimal digit, replace that 8 with the number to the right of the backslash in the leftmost column.

For example, the third row (2/A) under the 10_{16} column contains the symbol INTS*, representing the all integer subtract instructions. The opcode for those instructions would then be 12_{16} because the 0 in 10 is replaced by the 2 in the leftmost column. Likewise, the third row under the 18_{16} column contains the symbol JSR*, representing all jump instructions. The opcode for those instructions is 1A because the 8 in the heading is replaced by the number to the right of the backslash in the leftmost column.

The instruction format is listed under the instruction symbol.

The symbols in Table C-12 are explained in Table C-13.

Table C-12 • Opcode Summary

	00	08	10	18	20	28	30	38
0/8	PAL* (pal)	LDA (mem)	INTA* (op)	MISC* (mem)	LDF (mem)	LDL (mem)	BR (br)	BLBC (br)
1/9	Res	LDAH (mem)	INTL* (op)	\PAL\	LDG (mem)	LDQ (mem)	FBEQ (br)	BEQ (br)
2/A	Res	Res	INTS* (op)	JSR* (mem)	LDS (mem)	LDL_L (mem)	FBLT (br)	BLT (br)
3/B	Res	LDQ_U (mem)	INTM* (op)	\PAL\	LDT (mem)	LDQ_L (mem)	FBLE (br)	BLE (br)
4/C	Res	Res	Res	Res	STF (mem)	STL (mem)	BSR (br)	BLBS (br)
5/D	Res	Res	FLTV* (op)	\PAL\	STG (mem)	STQ (mem)	FBNE (br)	BNE (br)
6/E	Res	Res	FLTI* (op)	\PAL\	STS (mem)	STL_C (mem)	FBGE (br)	BGE (br)
7/F	Res	STQ_U (mem)	FLTL* (op)	\PAL\	STT (mem)	STQ_C (mem)	FBGT (br)	BGT (br)

Table C-13 • Key to Opcode Summary (Table C-12)

Symbol	Meaning
FLTI*	IEEE floating-point instruction opcodes
FLTL*	Floating-point Operate instruction opcodes
FLTV*	VAX floating-point instruction opcodes
INTA*	Integer arithmetic instruction opcodes
INTL*	Integer logical instruction opcodes
INTM*	Integer multiply instruction opcodes
INTS*	Integer subtract instruction opcodes
JSR*	Jump instruction opcodes
MISC*	Miscellaneous instruction opcodes
PAL*	PALcode instruction (CALL_PAL) opcodes
\PAL\	Reserved for PALcode
Res	Reserved for Digital

Index

A

Add instructions
  See also Floating-point Operate
  add longword, 4-22
  add quadword, 4-24
  add scaled longword, 4-23
  add scaled quadword, 4-25
ADDF instruction, 4-83
ADDG instruction, 4-83
ADDL instruction, 4-22
ADDQ instruction, 4-24
Address Space Match (ASM), virtual cache coherency, 5-4
Address Space Number (ASN), virtual cache coherency, 5-4
ADDS instruction, 4-84
ADDT instruction, 4-84
Aligned byte/word memory accesses, A-11
ALIGNED data objects, 1-8
Alignment
  atomic longword, 5-2
  atomic quadword, 5-2
  data considerations, A-6
  double-width data paths, A-1
  D_floating, 2-6
  F_floating, 2-4
  G_floating, 2-5
  instruction, A-2
  longword, 2-2
  memory accesses, A-11
  quadword, 2-2
  S_floating, 2-8
  T_floating, 2-9
Alpha architecture
  See also Conventions
  addressing, 2-1
  overview, 1-1
  porting operating systems to, 1-1
  programming implications, 5-1
  registers, 3-1
  security, 1-6
Alpha Privileged Architecture Library
  See PALcode
AND instruction, 4-36
Arithmetic instructions, 4-21
  See also specific arithmetic instructions
Arithmetic left shift instruction, using logical shift for, 4-35
Arithmetic traps
  Division by Zero, 4-60
  Inexact Result, 4-60
  Integer Overflow, 4-60
  Invalid Operation, 4-59
  Overflow, 4-60
  programming implications for, 5-20
  TRAPB instruction with, 4-105
  Underflow, 4-60
Atomic access, 5-2
Atomic operations
  accessing longword datum, 5-2
  accessing quadword datum, 5-2
  updating shared data structures, 5-6
  using load locked and store conditional, 5-7
Atomic sequences, A-17

B

BEQ instruction, 4-17
BGE instruction, 4-17
BGT instruction, 4-17
BIC instruction, 4-36
BIS instruction, 4-36
BLBC instruction, 4-17
BLBS instruction, 4-17
BLE instruction, 4-17
BLT instruction, 4-17
BNE instruction, 4-17
Boolean instructions, 4-35
  logical functions, 4-36
Boolean stylized code forms, A-14
bpt (PALcode) instruction, 9-1,
BPT (PALcode) instruction, 8-1,

BR instruction, 4-18
Branch instruction format, 3-9
Branch instructions, 4-16
  See also Control instructions
  backward conditional, 4-17
  conditional branch, 4-17
  displacement, 4-17
  floating-point, summarized, 4-74
  forward conditional, 4-17
  opcodes for, C-2
  unconditional branch, 4-18
Branch prediction model, 4-15
Branch prediction stack, with BSR instruction, 4-18
BSR instruction, 4-18
bugchk (PALcode) instruction, 9-1,
BUGCHK (PALcode) instruction, 8-1
Byte data type, 2-1
Byte manipulation instructions, 4-41
  See also Extract instructions; Insert instructions; Mask instructions

C

/C opcode qualifier
  IEEE floating-point, 4-56
  VAX floating-point, 4-56
Cache coherency, 5-1, 5-19
  in multiprocessor environment, 5-5
Caches
  design considerations, A-1
  I-stream considerations, A-4
  MB and IMB instructions with, 5-19
  requirements for, 5-4
  Translation Buffer conflicts, A-8
  with powerfail/recovery, 5-4
callsys (PALcode) instruction, 9-1
CALL_PAL (Call Privileged Architecture Library) instruction, 4-100
Canonical form, 4-61
CFLUSH (PALcode) instruction, 8-8
Changed datum, 5-5
CHME (PALcode) instruction, 8-1
CHMK (PALcode) instruction, 8-1
CHMS (PALcode) instruction, 8-2
CHMU (PALcode) instruction, 8-2
Clear a register, A-13
CMOVEQ instruction, 4-37
CMOVGE instruction, 4-37
CMOVGT instruction, 4-37
CMOVLBC instruction, 4-37
CMOVLBS instruction, 4-37
CMOVLE instruction, 4-37
CMOVLT instruction, 4-37
CMOVNE instruction, 4-37
CMPBGE instruction, 4-42
CMPEQ instruction, 4-26
CMPGEQ instruction, 4-85
CMPGLE instruction, 4-85
CMPGLT instruction, 4-85
CMPLE instruction, 4-26
CMPLT instruction, 4-26
CMPTEQ instruction, 4-86
CMPTLE instruction, 4-86
CMPTLT instruction, 4-86
CMPTUN instruction, 4-86
CMPULE instruction, 4-27
CMPULT instruction, 4-27
Code forms, stylized, A-12
  Boolean, A-14
  load literal, A-13
  negate, A-14
  NOP, A-12
  NOT, A-14
  register, clear, A-13
  register-to-register move, A-14
Code sequences, A-11
Coherency
  cache, 5-1
  defined, 5-1
Compare instructions
  See also Floating-point Operate
  compare byte, 4-42
  compare integer signed, 4-26
  compare integer unsigned, 4-27
Conditional move instructions, 4-37
  See also Floating-point Operate
Console, overview, 7-1
Control instructions, 4-15

Conventions
code examples, 1-9
extents, 1-8
figures, 1-9
instruction format, 3-8
notation, 3-8
numbering, 1-6
ranges, 1-8
CPSY instruction, 4-78
CPSYN instruction, 4-78
CPYSE instruction, 4-78
CVTDG instruction, 4-89
CVTGD instruction, 4-89
CVTGF instruction, 4-89
CVTGQ instruction, 4-87
CVTLQ instruction, 4-79
CVTQF instruction, 4-88
CVTQG instruction, 4-88
CVTQL instruction, 4-79
CVTQS instruction, 4-91
CVTQT instruction, 4-91
CVTTQ instruction, 4-90
CVTTS instruction, 4-92

D

/D opcode qualifier
FPCR (Floating-point Control Register), 4-61
IEEE floating-point, 4-56
D-stream considerations, A-6
Data alignment, A-6
Data format, overview, 1-3
Data sharing (multiprocessor), A-7
synchonization requirement, 5-5
Data stream
See D-stream
Data types
byte, 2-1
IEEE floating-point, 2-6
longword, 2-2
longword integer, 2-9
quadword, 2-2
quadword integer, 2-10
unsupported in hardware, 2-11
VAX floating-point, 2-3
word, 2-1
Denormal, defined for floating-point, 4-54
Dirty zero, defined for floating-point, 4-54
DIVF instruction, 4-93
DIVG instruction, 4-93
Division
integer, A-12
performance impact of, A-12
DIVS instruction, 4-94
DIVT instruction, 4-94
DRAINA (PALcode) instruction, 8-8
Dual-issue instruction considerations, A-2
D_floating data type, 2-5
alignment of, 2-6
mapping, 2-5
restricted, 2-6

E

EQV instruction, 4-36
Exception handlers, B-2
TRAPB instruction with, 4-105
EXTBL instruction, 4-44
EXTLH instruction, 4-44
EXTLL instruction, 4-44
EXTQH instruction, 4-44
EXTQL instruction, 4-44
Extract instructions (list), 4-44
EXTWH instruction, 4-44
EXTWL instruction, 4-44

F

FBEQ instruction, 4-75
FBGE instruction, 4-75
FBGT instruction, 4-75
FBLE instruction, 4-75
FBLT instruction, 4-75
FBNE instruction, 4-75
FCMOVEQ instruction, 4-80
FCMOVGE instruction, 4-80
FCMOVGT instruction, 4-80
FCMOVLE instruction, 4-80
FCMOVLT instruction, 4-80

FCMOVNE instruction, 4-80
FETCH (Prefetch Data) instruction, 4-101
  performance optimization, A-10
FETCH_M (Prefetch Data, Modify Intent) instruction, 4-101
  performance optimization, A-10
Finite number, Alpha, contrasted with VAX, 4-54
Floating-point branch instructions, 4-74
Floating-point Control Register (FPCR), 4-61
  accessing, 4-63
  at processor initialization, 4-63
  bit descriptions, 4-62
  instructions to read/write, 4-82
  Operate instructions that use, 4-76
  saving and restoring, 4-64
Floating-point Convert instructions, 3-12
Floating-point division, performance impact of, A-12
Floating-point format, number representation (encodings), 4-55
Floating-point instructions
  Branch (list), 4-74
  faults, 4-53
  introduced, 4-53
  Memory format (list), 4-65
  Operate (list), 4-76
  rounding modes, 4-55
  terminology, 4-54
  trapping modes, 4-57
  traps, 4-53
Floating-point load instructions, 4-65
  load F_floating, 4-66
  load G_floating, 4-67
  load S_floating, 4-68
  load T_floating, 4-69
  with nonfinite values, 4-65
Floating-point operate instructions, 4-76
  add (IEEE), 4-84
  add (VAX), 4-83
  compare (IEEE), 4-86
  compare (VAX), 4-85
  conditional move, 4-80
  convert IEEE floating to IEEE floating, 4-92
  convert IEEE floating to integer, 4-90
  convert integer to IEEE floating, 4-91
  convert integer to integer, 4-79
  convert integer to VAX floating, 4-88
  convert VAX floating to integer, 4-87
  convert VAX floating to VAX floating, 4-89
  copy sign, 4-78
  divide (IEEE), 4-94
  divide (VAX), 4-93
  format of, 3-11
  move from/to FPCR, 4-82
  multiply (IEEE), 4-96
  multiply (VAX), 4-95
  opcodes for, C-3
  subtract (IEEE), 4-98
  subtract (VAX), 4-97
Floating-point registers, 3-2
Floating-point rounding modes
  IEEE, 4-56
  VAX, 4-56
Floating-point single-precision operations, 4-61
Floating-point store instructions, 4-65
  store F_floating, 4-70
  store G_floating, 4-71
  store S_floating, 4-72
  store T_floating, 4-73
  with nonfinite values, 4-65
Floating-point support
  FPCR (Floating-point Control Register), 4-61
  IEEE, 2-6
  IEEE standard 754-1985, 4-64
  instruction overview, 4-53
  longword integer, 2-10
  Operate instructions, 4-76
  optional with Alpha, 4-2
  quadword integer, 2-10
  rounding modes, 4-55
  single-precision operations, 4-61
  trap modes, 4-57
  VAX, 2-3

Floating-point trapping modes, 4-57
See also Arithmetic traps
imprecision from pipelining, 4-58
FPCR (Floating-point Control Register)
See Floating-point Control Register (FPCR)
F_floating data type, 2-3
alignment of, 2-4
compared to IEEE S_floating, 2-8
MAX/MIN, 4-55
operations, 4-61

G

gentrap (PALcode) instruction, 9-1
GENTRAP (PALcode) instruction, 8-2
G_floating data type, 2-4
alignment of, 2-5
mapping, 2-4
MAX/MIN, 4-55

H

halt (PALcode) instruction, 9-2
HALT (PALcode) instruction, 6-4, 8-8

I

/I opcode qualifier, IEEE floating-point, 4-58
I-stream
design considerations, A-2
modifying physical, 5-5
modifying virtual, 5-4
PALcode with, 6-1
with caches, 5-4
IEEE convert-to-integer trap mode, instruction notation for, 4-58
IEEE floating-point
See also Floating-point instructions
exception handlers, B-2
format, 2-6
FPCR (Floating-point Control Register), 4-61
hardware support, B-1
NaN, 2-6
options, B-1
standard charts, B-9
standard, mapping to, B-3
S_floating, 2-7
trap handling, B-4
trap modes, 4-58
T_floating, 2-8
IEEE floating-point instructions
add instructions, 4-84
compare instructions, 4-86
convert from integer instructions, 4-91
convert IEEE floating format instructions, 4-92
convert to integer instructions, 4-90
divide instructions, 4-94
multiply instructions, 4-96
opcodes for, C-3
Operate instructions, 4-76
qualifiers, summarized, C-3
subtract instructions, 4-98
IEEE rounding modes, 4-56
IEEE standard
conformance to, B-1
mapping to, B-3
support for, 4-64
IEEE trap modes, required instruction notation, 4-58
IGN (Ignore), 1-8
imb (PALcode) instruction, 9-1
IMB (PALcode) instruction, 5-16, 6-5, 8-2
virtual I-cache coherency, 5-5
IMP (Implementation Dependent), 1-9
Infinity, defined for floating-point, 4-54
INSBL instruction, 4-47
Insert instructions (list), 4-47
INSLH instruction, 4-47
INSLL instruction, 4-47
INSQH instruction, 4-47
INSQHIL (PALcode) instruction, 8-2
INSQHILR (PALcode) instruction, 8-3
INSQHIQ (PALcode) instruction, 8-3
INSQHIQR (PALcode) instruction, 8-3
INSQL instruction, 4-47
INSQTIL (PALcode) instruction, 8-3
INSQTILR (PALcode) instruction, 8-3
INSQTIQ (PALcode) instruction, 8-4
INSQTIQR (PALcode) instruction, 8-4

INSQUEL (PALcode) instruction, 8-4
INSQUEQ (PALcode) instruction, 8-4
Instruction encodings
  floating-point format, C-3
  summarized, C-1
Instruction formats
  Branch, 3-9
  conventions, 3-8
  Floating-point Convert, 3-12
  Floating-point operate, 3-11
  Memory, 3-8
  Memory jump, 3-9
  operand values, 3-8
  operands, 3-8
  Operate, 3-10
  operators, 3-5
  overview, 1-4
  PALcode, 3-12
  registers, 3-1
Instruction overview, 1-4
Instruction set
  See also Floating-point instructions; PALcode instructions
  access type field, 3-4
  Boolean (list), 4-35
  branch (list), 4-16
  byte (list), 4-41
  conditional move (integer), 4-37
  data type field, 3-5
  extract (list), 4-41
  floating-point subsetting, 4-2
  insert (list), 4-41
  integer arithmetic (list), 4-21
  introduced, 1-6
  jump (list), 4-16
  load memory integer (list), 4-4
  mask (list), 4-41
  miscellaneous (list), 4-99
  name field, 3-4
  opcode qualifiers, 4-3
  operand notation, 3-4
  overview, 4-1
  shift, arithmetic, 4-40
  shift, logical, 4-39
  software emulation rules, 4-2
  store memory integer (list), 4-4
  VAX compatibility, 4-106
Instruction stream
  see I-stream
INSWH instruction, 4-47
INSWL instruction, 4-47
Integer arithmetic instructions
  See Arithmetic instructions
Integer division, A-12
Integer registers
  defined, 3-1
  R31 restrictions, 3-1

J

JMP instruction, 4-19
JSR instruction, 4-19
JSR_COROUTINE instruction, 4-19
Jump instructions, 4-16, 4-19
  See also Control instructions
  branch prediction logic, 4-20
  coroutine linkage, 4-20
  return from subroutine, 4-19
  unconditional long jump, 4-20

L

LDA instruction, 4-5
LDAH instruction, 4-5
LDF instruction, 4-66
LDG instruction, 4-67
LDL instruction, 4-6
LDL_L instruction, 4-8
  restrictions, 4-9
  with processor lock register/flag, 4-8
  with STx_C instruction, 4-8
LDQ instruction, 4-6
LDQP (PALcode) instruction, 8-8
LDQ_L instruction, 4-8
  restrictions, 4-9
  with processor lock register/flag, 4-8
  with STx_C instruction, 4-8
LDQ_U instruction, 4-7
LDS instruction, 4-68
LDT instruction, 4-69
Literals, operand notation, 3-4

Load instructions
See also Floating-point load instructions
emulation of, 4-2
FETCH instruction, 4-101
load address, 4-5
load address high, 4-5
load quadword, 4-6
load quadword locked, 4-8
load sign-extended longword, 4-6
load sign-extended longword locked, 4-8
load unaligned quadword, 4-7
multiprocessor environment, 5-5
serialization, 4-103
Load literal, A-13
Load memory integer instructions (list), 4-4
Location, 5-9
Location access order
defined, 5-11
with processor issue order, 5-12
Lock flag, per-processor
defined, 3-2
with load locked instructions, 4-8
with store conditional instructions, 4-11
Lock registers, per-processor
defined, 3-2
with load locked instructions, 4-8
with store conditional instructions, 4-11
Logical instructions
See Boolean instructions
Longword data type, 2-2
atomic access of, 5-2
integer floating-point format, 2-10
LSB (least significant bit), defined for
floating-point, 4-54

M

/M opcode qualifier, IEEE floating-point,
4-56
Mask instructions (list), 4-49
MAX, defined for floating-point, 4-55
MB (Memory Barrier) instruction, 4-103
See also IMB
multiprocessors only, 4-103
using, 5-17
with DMA I/O, 5-16
with multiprocessor D-stream, 5-16
MBZ (Must be Zero), 1-8
Memory access
aligned byte/word, A-11
coherency of, 5-1
granularity of, 5-2
width of, 5-2
Memory access sequence, 5-11
Memory alignment, requirement for, 5-2
Memory format instructions
function codes, summarized, C-1
opcodes for, C-1
Memory instruction format, 3-8
with function code, 3-9
Memory jump instruction format, 3-9
Memory management, support in PALcode,
6-1
Memory prefetch registers, A-10
defined, 3-2
Memory-like behavior, 5-3
MFPR (PALcode) instruction, 8-8
MF_FPCR instruction, 4-82
MIN, defined for floating-point, 4-55
Miscellaneous instructions (list), 4-99
Move instructions (conditional)
See Conditional move instructions
Move, register-to-register, A-14
MSKBL instruction, 4-49
MSKLH instruction, 4-49
MSKLL instruction, 4-49
MSKQL instruction, 4-49
MSKWH instruction, 4-49
MSKWL instruction, 4-49
MTPR (PALcode) instruction, 8-8
MT_FPCR instruction, 4-82
synchronization requirement, 4-63
MULF instruction, 4-95
MULG instruction, 4-95
MULL instruction, 4-28
with MULQ, 4-28
MULQ instruction, 4-29
with MULL, 4-28
with UMULH, 4-29
MULS instruction, 4-96
MULT instruction, 4-96

Multiple instruction issue, A-2
Multiply instructions
  See also Floating-point Operate
  multiply longword, 4-28
  multiply quadword, 4-29
  multiply unsigned quadward high, 4-30
Multiprocessor environment
  See also Data sharing
  cache coherency in, 5-5
  context switching, 5-17
  I-stream reliability, 5-16
  MB instruction, 5-16
  no implied barriers, 5-15
  read/write ordering, 5-8
  serialization requirements in, 4-103
  shared data, 5-5, A-7

N

NaN (Not-a-Number)
  defined, 2-6
  Quiet, 4-54
  Signaling, 4-54
NATURALLY ALIGNED data objects
  See ALIGNED data objects
Negate stylized code form, A-14
Non-memory-like behavior, 5-3
NOP, A-12
NOT instruction, ORNOT with zero, 4-36
NOT stylized code form, A-14

O

Opcode qualifiers
  See also specific qualifiers
  default values, 4-3
  notation (list), 4-3
Opcodes
  reserved, C-6
  summarized, C-6
Operand expressions, 3-3
Operand notation
  defined, 3-2
  from VAX architecture standard, 3-4
Operand values, 3-3
Operate format instructions, opcodes for, C-2
Operate instruction format, 3-10
  Floating-point, 3-11
  Floating-point Convert, 3-12
Operators, instruction format, 3-5
Optimization
  See Performance optimizations
ORNOT instruction, 4-36
OSF/1 privileged PALcode instructions, 9-2
OSF/1 unprivileged PALcode instructions, 9-1

P

PALcode
  barriers with, 5-15
  CALL_PAL instruction, described, 4-100
  compared to hardware instructions, 6-1
  Digital-defined for Alpha OSF/1, 9-1
  Digital-defined for Alpha VMS, 8-1
  implementation-specific, 6-1
  instead of microcode, 6-1
  instruction format, 3-12
  overview, 6-1
  privileged Alpha OSF/1, 9-2
  privileged VAX VMS, 8-8
  replacing, 6-2
  required function support, 6-2
  required instructions, 6-3
  running environment, 6-1
  special functions, 6-2
  unprivileged Alpha OSF/1, 9-1
  unprivileged Alpha VMS, 8-1
PALcode instructions
  opcodes for required, C-5
  opcodes reserved for, C-6
PALRES0, 6-2
PALRES1, 6-2
PALRES2, 6-2
PALRES3, 6-2
PALRES4, 6-2
PC
  See Program Counter register
PCC
  See Process Cycle Counter

Performance optimizations
branch prediction, A-3
code sequences, A-11
D-stream, A-6
for frequently executed code, A-1
for I-streams, A-2
I-stream density, A-4
instruction alignment, A-2
instruction scheduling, A-5
multiple instruction issue, A-2
shared data, A-7
Prefetch data (FETCH instruction), 4-101
Prefetch data registers, A-10
Prefetching data, considerations, A-10
Privileged Architecture Library
See PALcode
PROBE (PALcode) instruction, 8-4
Process Cycle Counter (PCC), RPCC instruction with, 4-104
Processor issue order
defined, 5-10
with location access order, 5-12
Processor issue sequence, 5-10
Program Counter (PC) register, 3-1
Pseudo-ops, A-15

Q

Quadword data type, 2-2
alignment of, 2-2
atomic access of, 5-2
integer floating-point format, 2-10
T_floating with, 2-10

R

R31, restrictions, 3-1
RAZ (Read as Zero), 1-8
RC (Read and Clear) instruction, 4-107
rdps (PALcode) instruction, 9-2
rdunique (PALcode) instruction, 9-1
rdusp (PALcode) instruction, 9-2
rdval (PALcode) instruction, 9-2
RD_PS (PALcode) instruction, 8-4
Read/write ordering (multiprocessor), 5-8
determining requirements, 5-8
memory location defined, 5-9
Read/write, sequential, A-9
READ_UNQ (PALcode) instruction, 8-4
Register-to-register move, A-14
Registers, 3-1
floating-point, 3-2
integer, 3-1
lock, 3-2
memory prefetch, 3-2
optional, 3-2
Program Counter (pc), 3-1
value when unused, 3-8
VAX compatibility, 3-2
REI (PALcode) instruction, 8-5
REMQHIL (PALcode) instruction, 8-5
REMQHILR (PALcode) instruction, 8-5
REMQHIQ (PALcode) instruction, 8-5
REMQHIQR (PALcode) instruction, 8-5
REMQTIL (PALcode) instruction, 8-6
REMQTILR (PALcode) instruction, 8-6
REMQTIQ (PALcode) instruction, 8-6
REMQTIQR (PALcode) instruction, 8-6
REMQUEL (PALcode) instruction, 8-6
REMQUEQ (PALcode) instruction, 8-7
Representative result, defined for floating-point, 4-54
Reserved instructions, opcodes for, C-6
Reserved operand, defined for floating-point, 4-55
Result latency, A-5
RET instruction, 4-19
retsys (PALcode) instruction, 9-2
Rounding modes
See Floating-point rounding modes
RPCC (Read Process Cycle Counter) instruction, 4-104
RS (Read and Set) instruction, 4-107
RSCC (PALcode) instruction, 8-7
rti (PALcode) instruction, 9-2

S

/S opcode qualifier
IEEE floating-point, 4-58
VAX floating-point, 4-57
S4ADDL instruction, 4-23
S4ADDQ instruction, 4-25
S4SUBL instruction, 4-32
S4SUBQ instruction, 4-34
S8ADDL instruction, 4-23
S8ADDQ instruction, 4-25
S8SUBL instruction, 4-32
S8SUBQ instruction, 4-34
SBZ (Should be Zero), 1-8
Security holes, 1-6
with UNPREDICTABLE results, 1-7
Sequential read/write, A-9
Serialization, MB instruction with, 4-103
Shared data (multiprocessor), A-7
changed vs. updated datum, 5-5
Shared data structures
atomic update, 5-6
ordering considerations, 5-7
using Memory Barrier (MB) instruction, 5-8
Shared memory
access sequence, 5-10
accessing, 5-10
defined, 5-9
issue sequence, 5-10
Shift arithmetic instructions, 4-40
Shift logical instructions, 4-39
Single-precision floating-point, 4-61
SLL instruction, 4-39
Software considerations, A-1
See also Performance optimizations
SRA instruction, 4-40
SRL instruction, 4-39
STF instruction, 4-70
STG instruction, 4-71
STL instruction, 4-13
STL_C instruction, 4-11
with LDx_L instruction, 4-11
with processor lock register/flag, 4-11
Store instructions
See also Floating-point store instructions
emulation of, 4-2
FETCH instruction, 4-101
multiprocessor environment, 5-5
serialization, 4-103
store longword, 4-13
store longword conditional, 4-11
store quadword, 4-13
store quadword conditional, 4-11
store unaligned quadword, 4-14
Store memory integer instructions (list), 4-4
STQ instruction, 4-13
STQP (PALcode) instruction, 8-8
STQ_C instruction, 4-11
with LDx_L inst., 4-11
with processor lock register/flag, 4-11
STQ_U instruction, 4-14
STS instruction, 4-72
STT instruction, 4-73
SUBF instruction, 4-97
SUBG instruction, 4-97
SUBL instruction, 4-31
SUBQ instruction, 4-33
SUBS instruction, 4-98
SUBT instruction, 4-98
Subtract instructions
See also Floating-point Operate
subtract longword, 4-31
subtract quadword, 4-33
subtract scaled longword, 4-32
subtract scaled quadword, 4-34
SWASTEN (PALcode) instruction, 8-7
swpctx (PALcode) instruction, 9-2
SWPCTX (PALcode) instruction, 9-2
swpipl (PALcode) instruction, 9-2
S_floating data type
alignment of, 2-8
compared to F_floating, 2-8
exceptions, 2-8
format, 2-7
mapping, 2-7
MAX/MIN, 4-55
operations, 4-61

T

tbi (PALcode) instruction, 9-2
Timing considerations, atomic sequences, A-17
Trap handler, with non-finite arithmetic operands, 4-59
Trap handling, IEEE floating-point, B-4
Trap modes
  Floating-point, 4-57
  IEEE, 4-58
  IEEE convert-to-integer, 4-58
  VAX, 4-57
  VAX convert-to-integer, 4-58
Trap shadow
  defined, 4-58
  defined for floating-point, 4-55
  trap handler requirement for, 4-58
TRAPB (Trap Barrier) instruction, A-14
  described, 4-105
  with MT_FPCR, 4-63
  with trap shadow, 4-58
True result, defined for floating-point, 4-54
True zero, defined for floating-point, 4-54
T_floating data type
  alignment of, 2-9
  exceptions, 2-9
  format, 2-9
  MAX/MIN, 4-55

U

/U opcode qualifier
  IEEE floating-point, 4-58
  VAX floating-point, 4-57
UMULH instruction, 4-30
  with MULQ, 4-29
UNALIGNED data objects, 1-8
Unconditional long jump, 4-20
UNDEFINED results, 1-7
UNORDERED memory references, 5-8
UNPREDICTABLE results, 1-7
Updated datum, 5-5

V

/V opcode qualifier
  IEEE floating-point, 4-58
  VAX floating-point, 4-58
VAX compatibility instructions, restrictions for, 4-106
VAX compatibility register, 3-2
VAX convert-to-integer trap mode, 4-58
VAX floating-point
  See also Floating-point instructions
  D_floating, 2-5
  F_floating, 2-3
  G_floating, 2-4
  trap modes, 4-58
VAX floating-point instructions
  add instructions, 4-83
  compare instructions, 4 85
  convert from integer instructions, 4-88
  convert to integer instructions, 4-87
  convert VAX floating format instructions, 4-89
  divide instructions, 4-93
  multiply instructions, 4-95
  opcodes for, C-5
  Operate instructions, 4-76
  qualifiers, summarized, C-5
  subtract instructions, 4-97
VAX rounding modes, 4-56
VAX trap modes, required instruction notation, 4-58
VAX VMS privileged PALcode instructions, 8-8
Virtual D-cache, 5-3
  maintaining coherency of, 5-3
Virtual I-cache, 5-3
  maintaining coherency of, 5-5
VMS unprivileged PALcode instructions, 8-1

W

whami (PALcode) instruction, 9-3
Word data type, 2-1
wrent (PALcode) instruction, 9-3
wrfen (PALcode) instruction, 9-3
Write buffers, requirements for, 5-4
Write-back caches, requirements for, 5-4
WRITE_UNQ (PALcode) instruction, 8-7
wrkgp (PALcode) instruction, 9-3
wrunique (PALcode) instruction, 9-1
wrusp (PALcode) instruction, 9-3
wrval (PALcode) instruction, 9-3
wrvptptr (PALcode) instruction, 9-3
WR_PS_SW (PALcode) instruction, 8-7

X

XOR instruction, 4-36

Z

ZAP instruction, 4-52
ZAPNOT instruction, 4-52
Zero byte instructions (list), 4-52

Alpha Architecture Handbook

Reader's Comments

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (Format, accuracy, completeness, organization, etc.)

_____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

Does the publication satisfy your needs? _____

_____

_____

_____

What errors have you found? _____

_____

_____

_____

Additional Comments _____

_____

_____

_____

_____

Name _____

Title _____

Company _____

Address _____

City _____ State _____ Zip _____

(please tape here)

(please fold here)

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
85 Swanson Road (BXB1-1/F04)
Boxboro, MA 01719-9960

digital